UNIVERSIDAD POLITÉCNICA DE MADRID

ESCUELA TÉCNICA SUPERIOR
DE INGENIEROS DE TELECOMUNICACIÓN

ARQUITECTURA DE SOFTWARE PARA SISTEMAS DE TIEMPO REAL PARTICIONADOS

TESIS DOCTORAL

JOSÉ ANTONIO PULIDO PAVÓN
INGENIERO DE TELECOMUNICACIÓN
2007

DEPARTAMENTO DE INGENIERÍA DE SISTEMAS TELEMÁTICOS

ESCUELA TÉCNICA SUPERIOR
DE INGENIEROS DE TELECOMUNICACIÓN

ARQUITECTURA DE SOFTWARE PARA SISTEMAS DE TIEMPO REAL PARTICIONADOS

Autor: José Antonio Pulido Pavón
Ingeniero de Telecomunicación

Director: Juan Antonio de la Puente Alfaro
Doctor Ingeniero Industrial

2007

Tribunal nombrado por el Magfco. y Excmo. Sr. Rector de la Universidad Politécnica de Madrid, el día ___ de _____ de 200__.

Presidente: _____

Vocal: _____

Vocal: _____

Vocal: _____

Secretario: _____

Suplente: _____

Suplente: _____

Realizado el acto de defensa y lectura de la Tesis el día ___ de _____ de 200__ en la E.T.S.I. /Facultad _____.

Calificación:_____

EL PRESIDENTE LOS VOCALES

EL SECRETARIO

Resumen

Los sistemas de tiempo real críticos embarcados en misiones aeroespaciales deben afrontar unos requisitos de seguridad y fiabilidad extremadamente exigentes debido a las circunstancias especiales que los rodean. Dichos requisitos afectan tanto al *hardware*, como al *software*, provocando que su evolución sea muy lenta.

Tradicionalmente, a la hora de diseñar un sistema se distinguía claramente entre los componentes específicos de la misión y aquellos destinados a funciones de control, utilizando procesadores dedicados para aislar físicamente unos de otros y evitar así la propagación de fallos.

La entrada en el sector espacial de una nueva generación de procesadores más potentes permite alojar varias aplicaciones en un mismo nodo *hardware*. Hasta ahora se utilizan técnicas de planificación estáticas para gestionar los recursos. Dichas técnicas proporcionan un nivel de seguridad elevado pero su eficiencia no es especialmente alta por lo que, en parte, se está desaprovechando la capacidad de estos nuevos procesadores.

En esta tesis se propone una nueva arquitectura, basada en técnicas de planificación dinámicas que permiten aumentar la eficiencia de manera notable. Además, la arquitectura incluye una serie de técnicas de monitorización que palían los problemas de seguridad que por sí solas tienen las técnicas de planificación dinámicas, permitiendo que cumpla con los exigentes requisitos que tienen los sistemas empleados en misiones espaciales.

El diseño de la nueva arquitectura se ha realizado pensando en su integración posterior dentro de un modelo de componentes que facilite el proceso de desarrollo de productos nuevos, ahorrando costes y mejorando la productividad de la industria aeroespacial.

Abstract

High-Integrity real-time systems aimed to space missions must face stringent reliability, safety and security requirements, due to the especial conditions this kind of systems deal with. Such requirements have a strong influence on both hardware and software components, causing a slow progression of their capabilities.

Usually, payload components were clearly separated from those that perform control functions, using different dedicated computer boards for each function in order to build fault containment regions.

At the present time, a new generation of powerful processors has appeared in the space domain, allowing to host several applications in the same node. Up to now, static scheduling policies have been used to manage processor usage. This kind of techniques guarantee a high level of safety. However, they lack efficiency, and therefore, part of the power contributed by the new generation of processors is being under-utilized.

In this thesis a new architecture is presented, based on dynamic scheduling policies which allow to reach a high efficiency level. Moreover, this new architecture includes a set of monitoring techniques which cover the safety problems usually associated to dynamic policies, enhancing global reliability in order to fulfill the requirements attached to space missions.

The new architecture has been designed taking in mind its subsequent integration within a component model, which eases the development of new products while saving costs and improving productivity of the aerospace industry.

Agradecimientos

Un largo camino separa la primera matrícula en un curso de doctorado de la defensa de la tesis. Afortunadamente, no he estado solo en ningún momento. A lo largo de los cuatro años invertidos en completar el programa de doctorado siempre he tenido a mi lado personas que me han ayudado a superar los obstáculos y empujado a alcanzar la meta.

Me gustaría destacar la figura de mi director de tesis, Juan Antonio de la Puente, por ser la persona que me dio la oportunidad de adentrarme en el mundo de la investigación y la que me ha guiado de manera eficiente para lograr los objetivos marcados. Junto con él, quiero agradecer a los demás profesores del grupo, Alejandro Alonso, Juan Zamorano y Miguel Ángel de Miguel, tanto la ayuda técnica prestada, como el excelente ambiente que han sabido crear y mantener a lo largo de estos años, sin el cual todo habría resultado mucho más complicado.

Igualmente crucial ha sido la colaboración de todos mis compañeros de laboratorio, especialmente de los que han trabajado en el mismo proyecto que yo, Santiago Urueña y José Redondo, puesto que sin su colaboración este trabajo nunca habría visto la luz. Para todos los demás, Daniel Berjón, Daniel Tejera, Juan Pedro Silva, Javier Fernández y todos aquellos que en algún momento han formado parte del grupo, sólo tengo palabras de agradecimiento por generar ese estupendo ambiente que reina en el departamento.

Por último, y no por ello menos importante, debo expresar mi gratitud hacia toda mi familia (tanto de sangre como política) por haber contribuido en la medida de sus posibilidades a que yo pueda estar hoy escribiendo esta página de agradecimientos. A todos ellos y en especial a Natalia, por ser lo mejor que me ha pasado en este mundo, gracias.

Abreviaturas

AADL Architecture, Analysis and Design Language

CAP Contenedores a nivel de APlicación

CMV Contenedores a nivel de Máquina Virtual

DMS Deadline Monotonic Scheduling

EDF Earliest Deadline First

ESA European Space Agency

IMA Integrated Modular Avionics

FIFO First In, First Out

FPPS Fixed Priority Preemptive Scheduling

MMU Memory Management Unit

ORK Open Ravenscar real-time Kernel

RMS Rate Monotonic Scheduling

RTA Response Time Analysis

SPARC Scalable Processor ARChitecture

SRP Stack Resource Policy

STR Sistema de Tiempo Real

UML Unified Modeling Language

WCET Worst Case Execution Time

Índice general

Capítulo 1

INTRODUCCIÓN

1.1. Motivación

Durante la última década, el grupo de investigación en *Sistemas de Tiempo Real y Arquitectura de Servicios Telemáticos*, del que formo parte, ha colaborado con la Agencia Europea del Espacio (ESA) a través de proyectos de diversa índole. Durante este periodo se ha adquirido una experiencia muy valiosa acerca de temas relacionados con los sistemas de tiempo real críticos destinados a misiones espaciales. Sobre este dominio vamos a fijar nuestra atención a lo largo de esta tesis.

Los sistemas de tiempo real comparten una característica común, consistente en que no sólo deben proporcionar una respuesta correcta desde el punto de vista lógico, sino que además deben hacerlo atendiendo a unas restricciones temporales dadas. Además, el hecho de que un sistema vaya a ser utilizado en misiones espaciales implica la necesidad de cumplir con unos requisitos de seguridad y fiabilidad muy estrictos, pues hay que tener presente las circunstancias que rodean a este tipo de sistemas:

- No es posible probar su comportamiento en condiciones reales de funcionamiento antes de su puesta en marcha definitiva.
- La sustitución de un componente es prácticamente inviable.
- Los componentes *hardware* utilizados en misiones espaciales no son ordinarios, deben estar protegidos frente a radiación, lo cual implica la utilización de una serie de técnicas especiales que aumentan considerablemente su coste a la vez que disminuyen sus prestaciones.

Debido a los factores inherentes al dominio espacial, el *hardware* apto para ser embarcado en estas misiones evoluciona muy lentamente. Este hecho se traduce en que se dispone de recursos muy limitados, tanto la velocidad del procesador como la capacidad de memoria utilizadas en misiones espaciales están varios órdenes de magnitud por debajo de los valores típicos que se pueden encontrar en ordenadores personales de hoy en día.

Como consecuencia de las escasas prestaciones del *hardware*, y de las precauciones que se deben tomar con el *software*, dado el impacto que un fallo puede acarrear, las aplicaciones específicas de la misión (denominada carga de pago o *payload*) y aquellas que realizan funciones de control (sistemas de navegación, comunicaciones, control de potencia, etcétera), tradicionalmente se ejecutaban en procesadores dedicados, es decir, se utilizaba una técnica de aislamiento físico mediante la cuál, las aplicaciones se desplegaban sobre nodos diferentes.

Gracias a una nueva generación de procesadores (ERC32, LEON. . .), construidos conforme a la arquitectura SPARC, actualmente se dispone de plataformas *hardware* más potentes que los anteriores, aunque aún muy alejadas de las prestaciones de un ordenador personal corriente.

Si con estas nuevas plataformas se continuara utilizando la técnica del aislamiento físico se estarían desperdiciando recursos, ya que dichas plataformas tienen capacidad suficiente para alojar varias aplicaciones en un solo nodo. Por este motivo, el método de aislamiento físico está cada vez más en desuso, siendo sustituido por técnicas de particionado que permiten ejecutar varias aplicaciones en un mismo nodo. No obstante, hasta ahora la tendencia ha sido utilizar técnicas de planificación estáticas, que son muy seguras, pero alcanzan unos niveles de rendimiento discretos.

Para superar esta barrera y conseguir aprovechar las prestaciones de la nueva generación de procesadores, esta tesis propone utilizar técnicas de planificación concurrente dinámicas que permitan maximizar la eficiencia del sistema. Debe destacarse que la utilización de este tipo de técnicas también implica la aparición de nuevos problemas que podrían comprometer la seguridad del sistema si no se toman las medidas adecuadas. Por ello, es necesario dotarlo con un conjunto de funciones que permitan garantizar el aislamiento temporal entre las diferentes aplicaciones, estableciendo barreras de contención de fallos que eviten que, debido a un error, una aplicación se exceda en el uso del procesador e impida la correcta ejecución de alguna otra aplicación.

En definitiva, para mejorar la situación de los sistemas de tiempo real críticos embarcados en misiones espaciales se hace necesario un nuevo marco de desarrollo de *software*

que permita ejecutar varias aplicaciones en un mismo nodo de manera eficiente, sustituyendo técnicas estáticas por técnicas dinámicas, pero manteniendo el nivel de seguridad que a día de hoy ofrecen las técnicas estáticas, cuyo uso está muy extendido en el sector.

1.2. Objetivos

Tras evaluar la situación descrita en el apartado anterior, estimamos que existe un apreciable margen de mejora en el rendimiento de los sistemas de tiempo real críticos embarcados en misiones aeroespaciales. Por tanto, el objetivo global de esta tesis consiste en diseñar una arquitectura de *software* para sistemas de tiempo real particionados, basada en técnicas de planificación dinámicas, orientada a mejorar la eficiencia obtenida mediante los métodos utilizados actualmente, y que contenga los mecanismos de monitorización necesarios para cumplir con los requisitos de seguridad exigidos en este tipo de misiones.

Para alcanzar y validar este objetivo principal se ha dividido el trabajo en varias etapas, estableciéndose la siguiente lista de objetivos parciales como los pasos fundamentales que se deben ir dando para cumplir con el objetivo global:

- Estudiar los métodos más relevantes utilizados actualmente para proporcionar aislamiento temporal, analizando sus debilidades y fortalezas.
- Proponer un método basado en un algoritmo dinámico que aproveche las ventajas de los anteriores a la vez que minimice sus inconvenientes.
- Analizar los elementos novedosos incorporados en la nueva revisión del lenguaje Ada, denominada Ada 2005.
- Valorar la utilidad de las nuevas funciones presentes en Ada 2005 para proporcionar aislamiento temporal.
- Desarrollar una plataforma de ejecución capaz de soportar las nuevas funciones mencionadas.
- Proponer soluciones basadas en los puntos anteriores que sean fácilmente integrables en un modelo de componentes.
- Validar las soluciones propuestas en esta tesis en el marco de la industria espacial europea, mediante proyectos piloto que hagan uso de las nuevas técnicas aportadas con objeto de evaluar su rendimiento en entornos reales.

- Divulgar los avances a través de congresos científicos internacionales de reconocido prestigio dentro del área técnica correspondiente.

1.3. Proyectos relacionados

Uno de los aspectos que más se ha cuidado a la hora de elaborar esta tesis es su integración dentro de un marco de investigación más completo, que aborde líneas de acción más amplias que las consideradas en este trabajo. De esta manera, se garantiza que se parte de una base técnica sólida y que el resultado final será fácilmente integrable en entornos más generales, evitando que se quede aislado debido a la dificultad de conectarlo con otros trabajos relacionados. Así pues, podemos resaltar varios enlaces de esta tesis con proyectos nacionales e internacionales (brevemente descritos a continuación), así como con un gran número de artículos científicos publicados en congresos o revistas de prestigio cuyas referencias irán apareciendo paulatinamente a lo largo de esta memoria.

1.3.1. El núcleo ORK

Tras una larga colaboración, a través de varios proyectos, con la *Agencia Europea del Espacio*, el *grupo de sistemas de tiempo real y arquitectura de servicios telemáticos* ha desarrollado un núcleo de tiempo real denominado ORK *(Open Ravenscar real-time Kernel)*.

ORK es un núcleo de tamaño mínimo, diseñado conforme al perfil de Ravenscar. Como se explicará a lo largo de la sección 2.3, someterse a las restricciones del perfil de Ravenscar supone eliminar todas aquellas construcciones que producen indeterminismo temporal o necesitan un soporte muy complejo, de esta manera es posible elaborar un núcleo de tamaño reducido, que aumenta la velocidad de ejecución de las aplicaciones y permite que el comportamiento temporal de éstas sea predecible.

ORK ha servido como punto de partida en la parte de la investigación relacionada con la plataforma de ejecución, puesto que se ha utilizado este núcleo para implementar los servicios que monitorizan el consumo de tiempo de ejecución exigidos por la nueva arquitectura.

1.3.2. El proyecto ASSERT

El proyecto ASSERT[1], co-financiado por la Comunidad Europea dentro del *VI Programa Marco* de cooperación entre los estados miembros, es un *proyecto integrado* coordinado por la Agencia Espacial Europea en el que toman parte aproximadamente 30 entidades de diversos tipos (instituciones, universidades y empresas privadas procedentes de 10 estados miembros), entre las que podemos destacar Alcatel, EADS, DASSAULT, el ETH (*Swiss Federal Institute of Technology*), la Universidad de Padua y la propia Universidad Politécnica de Madrid.

El proyecto ASSERT está encuadrado en el área de *Tecnologías de la Sociedad de la Información*, y surgió para actuar en consecuencia con los desafíos socio-económicos actuales:

- Resolviendo problemas de seguridad y rendimiento para mejorar ostensiblemente la fiabilidad de las tecnologías, infraestructuras y aplicaciones.
- Dando soporte para resolver problemas complejos en las áreas de ciencia, sociedad, industria y negocios.

Dentro de ese área tan amplia, el proyecto tiene un extenso rango de objetivos, entre los que destacan los siguientes puntos de interés:

- Sistemas empotrados: conceptos, métodos y herramientas de diseño de sistemas, desarrollo de componentes *software* con garantías e implementación de sistemas, todo ello, haciendo énfasis en el manejo adecuado de las restricciones de tiempo real.
- Sistemas distribuidos: *middleware* y plataformas para construir sistemas empotrados en red eficientes y de bajo coste, haciendo hincapié en ocultar en la medida de lo posible la complejidad de las comunicaciones subyacentes.
- Competitividad: reforzar las áreas en las que la industria europea es actualmente líder, así como superar las debilidades en aquellas en las que actualmente el retraso compromete el crecimiento y la capacidad de superar los desafíos sociales.

El proyecto tiene una estructura global compleja, dividida en tres *clusters* principales, centrados respectivamente en los aspectos relacionados con:

[1] Automated proof based System and Software Engineering for Real-Time Applications

- Ingeniería de sistemas basada en demostraciones.
- Fiabilidad, distribución y tiempo real estricto.
- Tecnologías de desarrollo y verificación.

A su vez, cada uno de estos *clusters* está estrechamente relacionado con otras estructuras del proyecto compuestas por expertos académicos (universidades e institutos tecnológicos) y socios industriales que se encargan de dar realimentación y evaluaciones sobre los trabajos desarrollados, así como de diseminar los resultados finales.

Nuestro grupo de investigación forma parte del segundo de los *clusters* mencionados en la lista anterior, siendo su objetivo principal el desarrollo de una plataforma que permita ejecutar redes de sistemas empotrados, capaz de ocultar la complejidad de las comunicaciones y de los cómputos inherentes a este tipo de sistemas, a la vez que provea una distribución efectiva y eficiente de los recursos.

La arquitectura propuesta en esta tesis será la base de la plataforma de ejecución desarrollada en el proyecto ASSERT, con lo que se tendrá la oportunidad de validar su funcionamiento en el entorno de la industria aeroespacial europea, siendo éste un excelente marco para comprobar la viabilidad de la propuesta y para contribuir a propagar hacia la industria los resultados de la investigación producida en el ámbito académico.

1.3.3. El proyecto TRECOM

El proyecto TRECOM (2002-2005) [2], es parte del *Plan nacional de I+D+I*. La realización fue llevada a cabo por tres grupos de investigación, que son:

- Grupo de sistemas de tiempo real y arquitectura de servicios telemáticos de la Universidad Politécnica de Madrid.
- Grupo de computadores y tiempo real de la Universidad de Cantabria.
- Grupo de informática industrial de la Universidad Politécnica de Valencia.

El objetivo del proyecto TRECOM fue desarrollar métodos y herramientas adecuados para la construcción de sistemas de tiempo real empotrados distribuidos con un alto grado de fiabilidad y requisitos de calidad de servicio, utilizando un enfoque de hacer un uso

[2] Sistemas de tiempo real empotrados, fiables y distribuidos basados en componentes

sistemático de tecnologías de componentes *software*, e integrar métodos de especificación y análisis de las propiedades relacionadas con la fiabilidad, la calidad de servicio y el comportamiento temporal de los sistemas. Entre los campos de aplicación estudiados se encuentran los de sistemas multimedia, sistemas ubicuos, y sistemas de control industrial, así como otros campos en que los requisitos de fiabilidad estrictos son comunes (sistemas de alta integridad).

Los resultados del proyecto comprenden: núcleos de sistemas operativos de tiempo real especializados para sistemas de alta integridad y gestión de calidad de servicio, subsistemas de comunicaciones y *software* intermedio para este tipo de sistemas, estudios sobre métodos de análisis de sistemas y herramientas de desarrollo y análisis adaptadas a la construcción de sistemas de tiempo real distribuidos.

Dentro del marco de esta tesis, en el proyecto TRECOM comenzó a desarrollarse la implementación de funciones de medida del tiempo de ejecución consumido por las diferentes tareas de un sistema, implementando sobre la versión de ORK disponible en aquel momento los relojes de tiempo de ejecución.

1.3.4. El proyecto THREAD

El proyecto THREAD (2005-2008) [3], es igualmente parte del *Plan nacional de I+D+I* y está siendo desarrollado por los mismos actores que el proyecto TRECOM.

En este caso, el objetivo global es el desarrollo de un soporte integral para sistemas empotrados, distribuidos y de tiempo real, en el que se incluirá una familia de plataformas interoperables, sus mecanismos de conexión, la arquitectura y las metodologías de diseño aplicables, y los dominios de aplicación de la próxima generación de este tipo de sistemas. Este soporte integral tendrá en cuenta todos los niveles desde el sistema operativo y las redes, pasando por el *middleware* de comunicaciones y de gestión de calidad de servicio, hasta llegar al nivel de la aplicación.

Los resultados que se espera obtener de la ejecución del proyecto comprenden plataformas configurables y abiertas que permitan la interoperabilidad entre distintos tipos de procesadores, sistemas operativos y lenguajes de programación, *software* intermedio (*middleware*) para sistemas empotrados distribuidos con requisitos de tiempo real y calidad de servicio, y directrices metodológicas y herramientas de desarrollo adecuadas para este tipo de sistemas.

[3] Soporte integral para sistemas empotrados de tiempo real distribuidos y abiertos

Dentro del proyecto THREAD se ha trabajado en la migración de la última versión de ORK, orientada a la plataforma *hardware* ERC32, a la plataforma *hardware* LEON (sucesora de ERC32), y en la monitorización avanzada del consumo de recursos.

1.4. Organización de la tesis

El resto del documento está organizado de la siguiente manera: el capítulo 2 hace un repaso de la teoría de sistemas de tiempo real (haciendo especial hincapié en la planificación), del lenguaje de programación Ada, ampliamente utilizado en el entorno de los STR, y de algunos lenguajes de modelado, que una vez consolidados en temas de propósito general también comienzan a penetrar en el ámbito del tiempo real. El capítulo 3 introduce el concepto de sistema particionado y repasa dos de los métodos más significativos a día de hoy, bien por su cuota de mercado (ARINC 653), bien por lo novedoso (servidores). En el capítulo 4 se presenta la principal contribución de esta tesis, una arquitectura de sistemas particionados basada en bandas de prioridad, mientras que en el capítulo 5 se enumeran los pasos necesarios para integrarla correctamente en el modelo de componentes definido en el proyecto ASSERT. El capítulo 6 resume el proceso que se ha seguido para validar la propuesta y, para finalizar, el capítulo 7 destaca las conclusiones extraídas de este trabajo.

Capítulo 2

ESTADO DE LA TÉCNICA

2.1. Sistemas de tiempo real

Existe una gran variedad de definiciones de sistema de tiempo real aunque todas ellas coinciden en algo fundamental: la relevancia del tiempo de respuesta. Una definición ampliamente aceptada es la que define un sistema de tiempo real como *"cualquier actividad o procesamiento de información que tiene que responder a un estímulo de entrada generado externamente (incluido el paso natural del tiempo), en un período finito y especificado"* (Young, 1982). Dicho de otra forma, en un sistema de tiempo real no basta con que la respuesta sea correcta desde el punto de vista lógico, sino que además ésta debe producirse dentro de un intervalo previamente especificado.

Se puede hacer una clasificación de los sistemas de tiempo real basándose en los requisitos temporales y las consecuencias que tendría la llegada de una respuesta fuera del tiempo previsto (Burns and Wellings, 2001). Así pues, se pueden distinguir tres categorías:

- Sistemas de tiempo real estricto: es absolutamente imprescindible que se cumplan siempre los plazos ya que una sola respuesta fuera del intervalo previsto podría tener consecuencias catastróficas, tales como la pérdida de vidas humanas o el fracaso total de la misión. Un claro ejemplo de STR estricto es el sistema ABS (*Sistema Anti-Bloqueo*) de un automóvil que evita que las ruedas queden bloqueadas durante la frenada (pues esto dejaría ingobernable la dirección del vehículo). Para ello, el sistema debe controlar la velocidad de giro de las ruedas y aflojar la presión que ejercen los frenos en caso de que se detecte un bloqueo inminente. Obviamente, una

respuesta tardía del sistema en esta situación podría desembocar en un accidente con víctimas.

- Sistemas de tiempo real flexible: el valor que tiene una respuesta decrece con el paso, del tiempo por lo que se permite que las respuestas lleguen fuera de plazo ocasionalmente. Por ejemplo, un sistema de control industrial que nivela el flujo a través de una tubería debe detectar cualquier desviación del nivel ideal y disparar una alarma lo antes posible. Cuanto más tarde el sistema en responder para corregir la desviación, mayores serán las pérdidas producidas.

- Sistemas de tiempo real firme: una respuesta tardía carece de valor, sin embargo, las consecuencias no son tan severas como en los sistemas estrictos por lo que pérdidas de plazos ocasionales son admisibles. Por ejemplo, en una planta de embotellado, si el grifo deja salir exactamente la cantidad de líquido previsto, pero lo hace unas décimas de segundo tarde, cuando la botella ya no está debajo, ésta se queda vacía y carece de valor, sin embargo, es admisible que este fallo suceda ocasionalmente siempre y cuando el porcentaje de errores se mantenga por debajo de un cierto umbral.

Por otra parte, es muy habitual que los STR formen parte de dispositivos que necesitan hacer algún tipo de procesado de información para cumplir con su objetivo principal, a pesar de que este último no tenga nada que ver con la informática, por ejemplo, podemos encontrar sistemas de tiempo real formando parte de automóviles, electrodomésticos, teléfonos, robots industriales, etcétera. Son lo que se denominan *sistemas empotrados* y al contrario de lo que pueda parecer a simple vista, los sistemas empotrados son tan habituales que se calcula que hasta el 99 % de los procesadores del mundo forman parte de ellos. Precisamente por el hecho de formar parte de sistemas cuya finalidad no es el procesado de información, los STR suelen tener una serie de características muy restrictivas, ya que en general se cuenta con recursos limitados en lo que se refiere a procesador y memoria, los dispositivos de entrada y salida son especiales (no suele haber teclado ni pantalla), e incluso la arquitectura del sistema también suele diferir de la propia de un sistema estándar (figura 2.1). Por estas razones, en los sistemas de tiempo real la eficiencia es un factor clave.

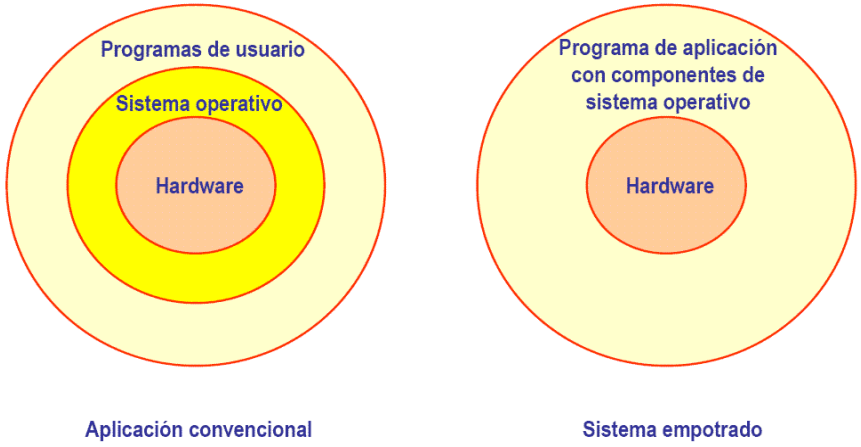

Figura 2.1: Arquitectura típica de un sistema empotrado

2.2. Planificación de sistemas de tiempo real

Dada la importancia que en los STR tiene la gestión del tiempo y la predictibilidad del comportamiento del sistema, es fundamental hacer una buena planificación para aprovechar al máximo los recursos disponibles (fundamentalmente el procesador) sin llegar a sobrecargarlos, evitando así que los trabajos se completen fuera del plazo previsto. Así pues, debe haber un elemento denominado *planificador* que se encargue de asignar los recursos apropiados a los trabajos pendientes para que se realicen de manera eficiente.

Esta labor resulta especialmente complicada si tenemos en cuenta que en la gran mayoría de sistemas (tanto mono como multiprocesador) se ejecutan varias aplicaciones simultáneamente haciendo uso de recursos compartidos (fundamentalmente procesador y memoria), dando lugar a la *concurrencia*.

A la hora de planificar un sistema concurrente utilizamos como unidad básica la *tarea*, que es una secuencia de operaciones que tiene restricciones de tiempo definidos a partir de una serie de parámetros (prácticamente equivalente a una hebra de ejecución de un sistema operativo). Una tarea se activa en un momento dado que llamamos *activación* y tiene que completar una serie de instrucciones que llevan asociado un cierto *tiempo de ejecución* antes de un instante que denominamos *plazo de respuesta* (a veces se utiliza la voz inglesa *deadline*). La figura 2.2 aclara estos conceptos. Dependiendo del modo en que se active la tarea, podemos hacer la siguiente clasificación:

- Tarea periódica: si se activa con un periodo fijo, marcado por el reloj del sistema.

- Tarea aperiódica: cuando la activación de la tarea, generalmente debida a algún evento externo, puede darse en cualquier instante.
- Tarea esporádica: si, de manera semejante a las tareas aperiódicas, se puede activar en cualquier instante pero respetando un intervalo mínimo entre dos activaciones consecutivas.

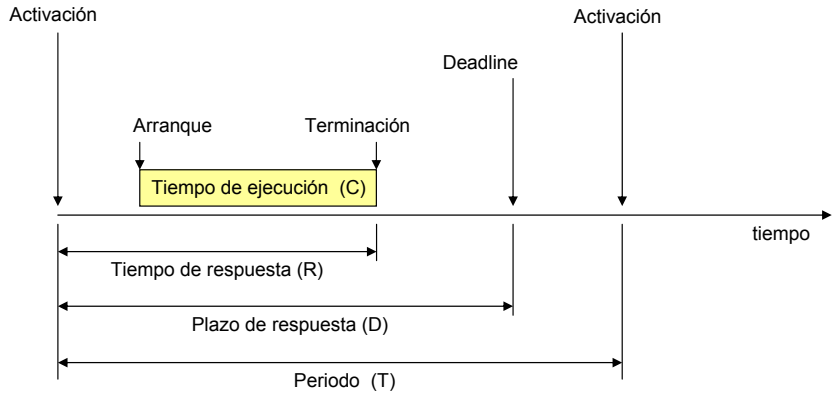

Figura 2.2: Parámetros de una tarea

A su vez, podemos distinguir entre tareas *no desalojables* y *desalojables* según el planificador deba asignarle un recurso a una tarea hasta que ésta finalice todas sus operaciones, o bien la pueda expulsar y diferir antes de completar su trabajo.

El problema de la planificación es muy complejo, ha sido ampliamente estudiado a lo largo de la historia de los sistemas de tiempo real (Sha et al., 2004) y, puesto que es uno de los fundamentos teóricos en los que se apoya esta tesis, los siguientes apartados se dedican a describir con detalle los métodos que enunciamos a continuación. Podemos identificar tres paradigmas fundamentales que engloban los diferentes métodos de planificación:

- Planificación estática: es un método síncrono, dirigido por el reloj del sistema. El tiempo se divide en rodajas que son asignadas a las diferentes tareas en tiempo de diseño, de manera que se conoce perfectamente a priori en qué momento se ejecutará cada tarea.
- Prioridades fijas: a cada tarea se le asigna una prioridad fija en tiempo de diseño que se mantiene invariable durante toda la vida de la tarea. El planificador se encarga

de que en cada instante ocupe el procesador aquella tarea que, estando lista para ser ejecutada, tenga una mayor prioridad asociada.

- Prioridades dinámicas: en este caso las tareas también tienen asignada una prioridad y el planificador elige en cada momento la más prioritaria, sin embargo, la prioridad de las tareas cambia dinámicamente en tiempo de ejecución de acuerdo con algún factor definido previamente, por ejemplo, la proximidad de la expiración de su plazo de respuesta.

2.2.1. Planificación estática

Inicialmente, la planificación estática fue el método más utilizado en los STR, siendo el enfoque del *ejecutivo cíclico* el más común (Baker and Shaw, 1989). La idea se basa en construir un sistema cíclico prácticamente hecho a medida. En el sistema pueden apreciarse una serie de ciclos secundarios de duración fija en los que se ejecutan ciertas tareas, cada una de ellas de principio a fin, no siendo necesario proteger los datos ya que este esquema no da lugar a concurrencia. Estos ciclos secundarios se agrupan dentro de un ciclo principal que se repite periódicamente tal y como muestra la figura 2.3, de ahí su nombre de ejecutivo cíclico.

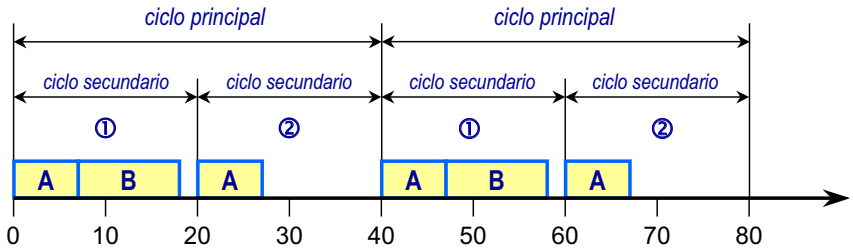

Figura 2.3: Ejemplo de ejecutivo cíclico

Este método es determinista por su propia naturaleza, ya que desde el diseño del sistema se conocen todos los detalles acerca de su posterior ejecución, por lo tanto, si es posible construir el ejecutivo cíclico no es necesario ir más allá en su análisis temporal porque el sistema es correcto por construcción.

La planificación estática es un método de planificación muy utilizado en los STR de alta integridad (Zamorano et al., 1997), ya que el determinismo es la característica más

valorada en términos de seguridad. Sin embargo, este método no está exento de problemas, pues su alto grado de determinismo es posible a costa de su escasa flexibilidad que, a su vez, se traduce en una serie de problemas. Por ejemplo, en el caso general, construir el ejecutivo cíclico es un problema NP-duro, que resulta difícil de resolver, especialmente si los periodos de las tareas son de diferentes órdenes de magnitud. Además, es muy costoso de mantener y reutilizar, ya que introducir una tarea nueva en un sistema existente implicaría rehacer toda su planificación. Asimismo, encajar las tareas aperiódicas, que pueden activarse en cualquier momento, dentro de este modelo que es totalmente síncrono tampoco es un problema trivial.

2.2.2. Prioridades fijas

La planificación con prioridades fijas es un procedimiento que impulsa la flexibilidad, transformando en dinámica parte de la gestión que en el método anterior se hacía de manera totalmente estática. La parte estática de la planificación basada en prioridades fijas consiste en establecer en tiempo de diseño una prioridad para cada tarea, que se mantendrá fija durante toda la vida de ésta. Posteriormente, ya en tiempo de ejecución, es el planificador el que se encarga de que en cada momento se ejecute la tarea con un nivel de prioridad más alto.

El trabajo de (Liu and Layland, 1973) fue la base de la teoría de planificación con prioridades fijas. En dicho trabajo se propone un protocolo de asignación de prioridades óptimo, conocido como RMS (*Rate Monotonic Scheduling*), por el cual se asignan los niveles de prioridad de manera inversamente proporcional a los periodos de las tareas, es decir, a la tarea que se activa con mayor frecuencia se le asigna la mayor prioridad y así sucesivamente. RMS es óptimo en el sentido de que si un conjunto de tareas es planificable con alguna asignación de prioridades, también lo será con RMS.

Este modelo inicial propuesto en 1973 era bastante limitado debido a que los autores impusieron una larga lista de restricciones al modelo computacional (todas las tareas independientes y periódicas, activadas al inicio de su periodo, con periodo igual a su plazo, . . .), sin embargo, a lo largo de los años han ido apareciendo estudios que eliminaban parte de las restricciones originales, haciendo que en la actualidad el modelo de planificación con prioridades fijas sea muy completo, potente y atractivo.

La planificación con prioridades fijas resuelve en gran parte los problemas que planteaba la planificación estática. Mientras que la construcción del ejecutivo cíclico era muy complicado, en este caso, asignar las prioridades, que es lo que se hace en tiempo de

diseño, es muy sencillo siguiendo esquemas como RMS. También es importante todo lo referido a la reasignación dinámica de recursos, ya que al tomar decisiones en tiempo de ejecución, hay cierto margen para corregir problemas derivados de un diseño incorrecto o algún imprevisto, como que el tiempo de ejecución de una tarea se alargue más de lo previsto.

La política de planificación mediante prioridades fijas permite aseverar que en caso de sobrecarga, aquellas tareas con un nivel de prioridad más bajo serán las primeras en incumplir su plazo de respuesta. Por tanto, si se decidiera asignar las prioridades en función de la importancia de las tareas, se podría afirmar que en caso de sobrecarga serían aquellas tareas menos importantes las primeras en perder su plazo.

A cambio de estas mejoras, el principal problema de la planificación con prioridades fijas es la pérdida de determinismo. Al no conocer el orden exacto en el que se ejecutarán las tareas ni el momento en el que se despacharán, se hacen necesarios nuevos métodos de análisis que permitan pronosticar si un sistema va a ser planificable o no.

El método de análisis más sencillo que nos informa acerca de la planificabilidad del sistema (esto es, si todas las tareas cumplirán sus plazos), es el denominado *factor de utilización del procesador*, que proporciona una condición suficiente (no necesaria), para que el sistema sea planificable. En el caso de utilizar RMS, la fórmula para calcularlo viene dada en la ecuación 2.1, siendo *N* el número de tareas que componen el sistema, C_i el tiempo de ejecución de la tarea *i* en el caso más desfavorable, y T_i su periodo de activación.

$$U(N) = \sum_{i=1}^{n} \frac{C_i}{T_i} \leq N \cdot (2^{\frac{1}{N}} - 1) \tag{2.1}$$

Lo más crítico de esta expresión es calcular el valor del término C_i, ya que determinar el peor tiempo de ejecución de una tarea no es sencillo, ya sea midiéndolo en una implementación piloto (difícil saber cuando se ejecuta el caso peor), o analizando su código (difícil de incluir todos los efectos provocados por el *hardware*), por lo que hay multitud de trabajos de investigación que proponen métodos para calcular dicho valor de una manera segura y eficiente (Bernat et al., 2002; Edgar and Burns, 2001). Por otra parte, se observa que a medida que crece el número de tareas *N*, el valor máximo de U(N) para el que los plazos están garantizados va decreciendo. Tomando límites se obtiene una cota inferior de ese valor:

$$\lim_{N \to \infty} [N \cdot (2^{\frac{1}{N}} - 1)] = \ln 2 \simeq 0{,}69$$

Un método de análisis más completo y flexible que el factor de utilización, aunque más complicado de llevar a cabo, es el *Análisis de Tiempo de Respuesta* (RTA), introducido a finales de la década de los 80 (Harter, 1987; Joseph and Pandya, 1986; Audsley et al., 1992). A diferencia del factor de utilización, el análisis de tiempo de respuesta establece una condición necesaria y suficiente para que el sistema sea planificable. Esta técnica se basa en comprobar que el tiempo de respuesta *R* de una tarea *i* es menor que su plazo correspondiente, es decir:

$$\forall i, R_i \leq D_i$$

Para ello, hay que calcular el tiempo de respuesta de cada tarea en el caso más desfavorable que, en el modelo de tareas más básico, viene dado por la ecuación recurrente 2.2, donde *hp(i)* representa el conjunto de tareas con prioridad mayor que *i*:

$$w_i^{n+1} = C_i + \sum_{j \in hp(i)} \left\lceil \frac{w_i^n}{T_j} \right\rceil C_j \tag{2.2}$$

La solución a esta ecuación se obtiene de manera recursiva, comenzando por hacer $w_i^n = C_i$ e iterando hasta que converja ($w_i^{n+1} = w_i^n$), y entonces $R_i = w_i^n$.

La ecuación 2.2 ha sido posteriormente modificada tras sucesivas aportaciones que han ido añadiendo nuevas características al modelo de tareas básico, como por ejemplo, inclusión de plazos de respuesta distintos de los periodos (Leung and Whitehead, 1982; Audsley et al., 1993), tareas aperiódicas (Sha et al., 1986; Sprunt et al., 1989; Strosnider et al., 1995), interferencias debidas a la exclusión mutua de los recursos compartidos (Sha et al., 1990), fluctuaciones (*jitters*) y fases (*offsets*) (Bate and Burns, 1997; Palencia and Harbour, 1998), resultando finalmente la ecuación 2.3, donde B_i representa el tiempo de bloqueo sufrido por la tarea *i* debido a la exclusión mutua en los recursos compartidos con otras tareas (ver explicación en la sección 2.2.4) y J_i es el *jitter*.

$$w_i^{n+1}(q) = (q+1)C_i + B_i + \sum_{j \in hp(i)} \left\lceil \frac{w_i^n(q) + J_j}{T_j} \right\rceil C_j \tag{2.3}$$

$$R_i(q) = w_i^n(q) - q \cdot T_i + J_i \tag{2.4}$$

donde finalmente:

$$R_i = \max_{q=1,2,...} R_i(q)$$

siendo el máximo valor de q que necesita ser probado el menor entero que verifica:

$$R_i(q) \leq T_i$$

Dado que un modelo de tareas completo dificulta extremadamente su análisis temporal, generalmente se recurre a incluir una serie de restricciones en el código, prohibiendo el uso de ciertas construcciones que generen indeterminismo temporal (ver sección 2.3.3). Así pues, es habitual hablar de un modelo de tareas estático en el que todas las tareas sean creadas en la elaboración del sistema, no terminen nunca y tengan un solo punto de activación.

2.2.3. Prioridades dinámicas

La planificación con prioridades dinámicas supone un nuevo aumento de flexibilidad, haciendo que un mayor número de decisiones se tomen dinámicamente, en tiempo de ejecución, en lugar de hacerlo estáticamente, en tiempo de diseño.

La diferencia fundamental entre la planificación con prioridades fijas, y la planificación con prioridades dinámicas radica en que, con el primer método, las prioridades se asignan estáticamente en función de algún parámetro de la tarea (periodo, plazo de respuesta, importancia, etc.) y permanecen ya constantes durante la ejecución. Sin embargo, cuando hablamos de prioridades dinámicas, la prioridad de cada tarea es una variable que va tomando diversos valores en tiempo de ejecución.

El método de planificación con prioridades dinámicas más extendido se conoce por las siglas EDF (*Earliest Deadline First*) y fue definido inicialmente por Liu y Layland en el artículo ya citado anteriormente (Liu and Layland, 1973). En este método, el parámetro que se utiliza para modificar el valor de la prioridad de las tareas es la proximidad de su plazo de respuesta, asignando las prioridades de manera directamente proporcional a dicha proximidad, es decir, a la tarea cuyo plazo está más cercano se le asigna la prioridad más alta.

La planificación EDF es óptima entre los algoritmos de planificación con desalojo, lo cual significa que si un conjunto de tareas es planificable mediante algún otro algoritmo, también lo es utilizando EDF.

Una vez más, la técnica de análisis más sencilla es la del factor de utilización. Además, en el caso de EDF nos da una condición que además de suficiente es necesaria, con lo cual, partiendo de las mismas premisas que en el apartado anterior cuando se utilizaba un conjunto de *N* tareas planificadas con RMS, en el caso de EDF podemos decir que es planificable si, y solo si, se verifica la ecuación 2.5:

$$U = \sum_{i=1}^{N} \frac{C_i}{T_i} \leq 1 \tag{2.5}$$

Comparando los ecuaciones 2.1 y 2.5 podemos ver fácilmente que para un número de tareas $N > 1$, la utilización que garantiza la planificabilidad del sistema es más alta en el caso de EDF. Es decir, teóricamente, con EDF se puede sacar más rendimiento del procesador. Ahora bien, esta ventaja teórica se ve seriamente mermada por dos desventajas prácticas:

- Si por algún motivo se produce una sobrecarga, esto es, la cantidad de tiempo de procesador restante es insuficiente para que no se pierda el plazo de alguna de las tareas activas, el comportamiento de la planificación EDF es impredecible.
- La complejidad de un planificador EDF es superior a la de uno basado en prioridades fijas, lo cual se traduce finalmente en mayores sobrecargas a la hora de gestionar los cambios de contexto.

2.2.4. Acceso a datos compartidos

En las secciones anteriores hemos pasado por alto el problema de los accesos a datos compartidos que puede haber en un sistema cuyas tareas no sean independientes. En estos casos, se hace necesario algún mecanismo de protección que evite que varias tareas accedan a la vez a un mismo dato de manera incontrolada, ya que esto podría provocar errores graves. La mayoría de los lenguajes incorporan algún tipo de mecanismo para solventar este problema (semáforos, monitores, objetos protegidos), que permiten que una tarea sea suspendida hasta que suceda algún evento que garantice que se puede acceder al recurso en cuestión con seguridad (por ejemplo, que se libere un semáforo).

El problema surge cuando las tareas que acceden a un recurso son de diferente prioridad (ver figura 2.4). En este ejemplo, las tareas τ_1 y τ_4 comparten un recurso. En un momento dado, la tarea τ_4 (de baja prioridad) se hace con él, cuando está a mitad de la ejecución en la zona compartida, se activan las tareas τ_2 y τ_3, que al ser más prioritarias,

la desalojan. Un poco más tarde se activa la tarea τ_1 que, al ser la más prioritaria comienza a ejecutarse inmediatamente, sin embargo, cuando intenta acceder al recurso compartido no puede hacerlo porque éste está ocupado por la tarea τ_4, así que se queda bloqueada esperando a que finalice esta ocupación, para lo que a su vez deben finalizar antes las tareas τ_2 y τ_3 también. El resultado final es que se ha producido una *inversión de prioridad*, ya que las tareas τ_2 y τ_3 han terminado su ejecución antes que la τ_1.

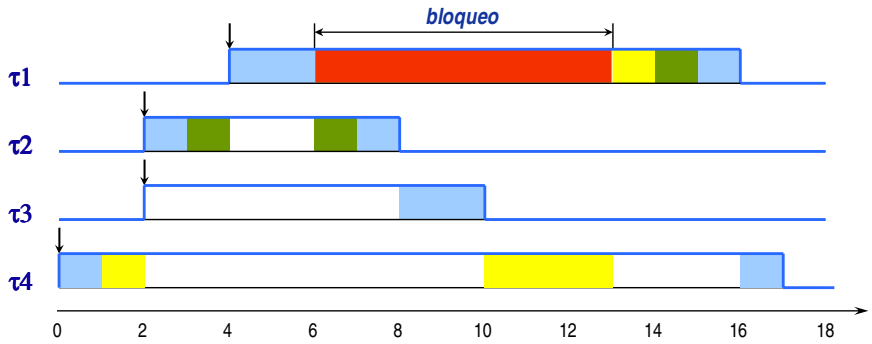

Figura 2.4: Inversión de prioridad

Esto que hemos explicado con prioridades fijas, sucede de manera análoga en la planificación EDF, donde el fenómeno se conoce como *inversión de tiempo límite*. En ambos casos se trata de un efecto perverso que no debe darse porque contraviene la esencia de las prioridades.

Si bien no es posible eliminar completamente los efectos secundarios de la inversión de prioridad, sí que se puede minimizar su impacto siguiendo ciertos protocolos que varían dinámicamente la prioridad de las tareas (Cornhilll et al., 1987). Por supuesto, esto supone una excepción en la teoría de prioridades fijas. Por tanto, cuando decíamos que en la planificación con prioridades fijas la prioridad de la tarea permanece invariable durante su ejecución, en realidad faltaba añadir: *"excepción hecha de cuando se modifica su prioridad para acceder a un recurso compartido"*. En los siguientes apartados presentamos los protocolos más relevantes (Rajkumar, 1991).

Protocolo de herencia de prioridad

La variante más sencilla que se puede poner en práctica para evitar la inversión de prioridad consiste en elevar la prioridad de la tarea que mantiene bloqueado un recurso compartido cuando una tarea de mayor prioridad intenta acceder a él (ver figura 2.5). Así,

cuando la tarea τ_1, intenta acceder al recurso ocupado, se eleva la prioridad de la tarea τ_4 para que desaloje el recurso cuanto antes (Cornhill and Sha, 1987). De esta manera se consigue evitar la inversión de prioridad.

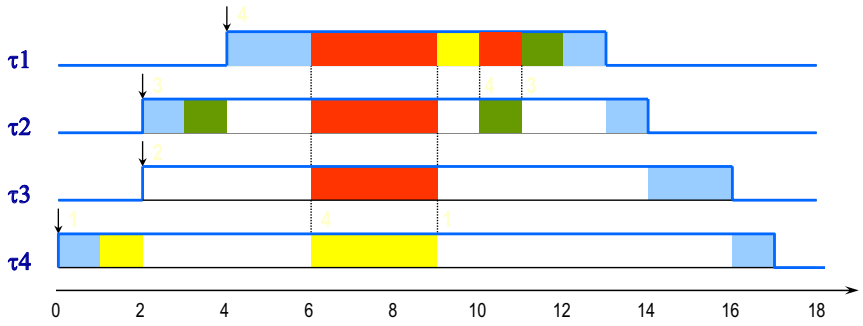

Figura 2.5: Protocolo de herencia de prioridad

Protocolo de techo de prioridad

El protocolo de techo de prioridad (Goodenough and Sha, 1988) es más complejo que el anterior. En este caso se define un techo de prioridad para cada recurso compartido. El valor de este techo es igual a la máxima prioridad de las tareas que acceden a él. Cada tarea tiene una prioridad dinámica que se calcula como el máximo entre la prioridad básica de la propia tarea y la prioridad de las tareas a las que bloquea. A partir de estas definiciones se establece que una tarea sólo puede acceder a un recurso compartido si su prioridad dinámica es mayor que el techo de prioridad de todos los recursos en uso.

En la figura 2.6 se puede observar cómo la tarea τ_1 comienza a ejecutarse hasta que es desalojada cuando se activan las tareas τ_2 y τ_3 de mayor prioridad. Sin embargo, cuando la tarea τ_2 intenta acceder a un recurso compartido no puede hacerlo al ser la prioridad dinámica de la tarea τ_1 mayor.

Comparando las figuras 2.5 y 2.6 se aprecia que el protocolo de techo de prioridad da un mejor resultado, pues hace que los recursos utilizados por tareas de alta prioridad sean liberados antes. Además de la anterior, podemos añadir algunas ventajas:

- Las tareas se bloquean como máximo una vez durante cada ciclo.
- No puede haber interbloqueos ni bloqueos encadenados.

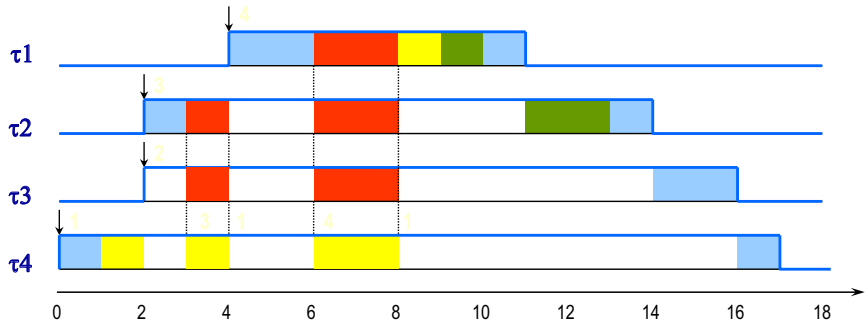

Figura 2.6: Protocolo de techo de prioridad

Protocolo de techo de prioridad inmediato

Siguiendo una filosofía similar al protocolo anterior, el método más eficiente de acceso a recursos compartidos en sistemas planificados con prioridades fijas es el protocolo de techo de prioridad inmediato (Sha et al., 1990).

En este caso, cuando una tarea accede a un recurso compartido inmediatamente se eleva su prioridad hasta el techo del recurso (es decir, a la máxima prioridad de todas las tareas que acceden a él). De esta manera se consigue que el tiempo de ocupación de un recurso compartido sea el más pequeño posible, minimizando así los tiempos de bloqueo.

El protocolo de techo de prioridad inmediato conserva las ventajas enumeradas para el protocolo de techo de prioridad pero, además, nos permite añadir una, y es que podemos afirmar que utilizando este protocolo, si una tarea se bloquea, siempre es al inicio de su ejecución, tal y como muestra la figura 2.7. En resumen, podríamos decir que a pesar de las semejanzas entre el protocolo de techo de prioridad (simple) y el protocolo de techo de prioridad inmediato, este último tiene dos ventajas fundamentales:

- Es más fácil de implementar.
- Provoca menos cambios de contexto.

Política de pila de recursos

En el caso de planificación EDF, existe un método propuesto por Baker (Baker, 1991) muy semejante al techo de prioridad, denominado *política de pila de recursos*, más co-

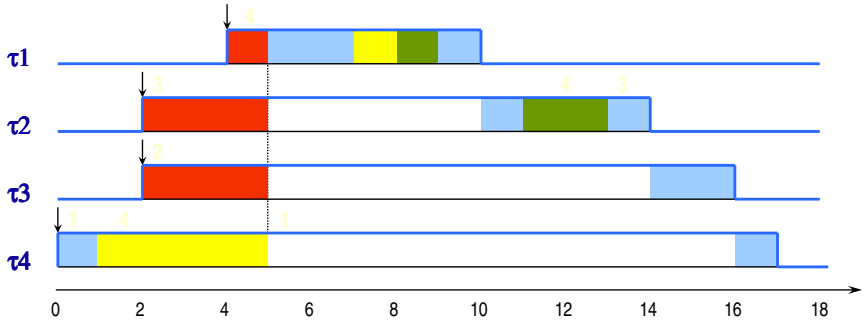

Figura 2.7: Protocolo de techo de prioridad inmediato

nocido por sus siglas en inglés: SRP (*Stack Resource Policy*). En este caso, a cada tarea se le asigna un nivel de apropiación en función de la cercanía de su plazo. En tiempo de ejecución a cada recurso compartido se le asigna una cota (semejante al techo de los protocolos vistos anteriormente), igual al nivel más alto de los niveles de apropiación que tienen las tareas que acceden a él. A partir de ahí se establece que cuando se active una tarea sólo podrá desalojar a la que está en el procesador si su plazo está más cercano y además, su nivel de apropiación supera a la cota de todos los recursos que hay bloqueados en el sistema en ese momento.

Con ambos protocolos se consigue que las tareas sufran como máximo un solo bloqueo, que además ocurre al ser activadas, se previenen bloqueos mutuos y se puede calcular fácilmente el tiempo que pueden permanecer bloqueados.

2.3. El lenguaje Ada y el perfil de Ravenscar

2.3.1. Generalidades del lenguaje Ada

Los orígenes del lenguaje de programación Ada datan aproximadamente de 1974, año en el que comenzó su desarrollo por parte del Departamento de Defensa de los Estados Unidos bajo la premisa de reducir los costes destinados a *software*, especialmente en lo relativo a sistemas de alta integridad y sistemas empotrados. El proceso de definición, desarrollo y pruebas dio finalmente sus frutos en el año 1983, cuando fue publicado el estándar ANSI para el lenguaje Ada bajo el título *Reference Manual for the Ada Programming Language* (ANS, 1983), siendo esta versión conocida popularmente como *Ada*

83. En 1987 se convirtió en un estándar ISO (ISO, 1987), aunque se mantuvo la denominación Ada 83.

Años más tarde, la constante evolución tecnológica llevó a plantearse la necesidad de revisar el estándar *Ada 83* con objeto de adaptarlo a los nuevos tiempos, aplicando la experiencia adquirida en entornos fuera de los sistemas empotrados para los que inicialmente se pensó el lenguaje y en nuevos paradigmas como la programación orientada a objetos. El nuevo resultado fue un estándar ISO publicado en 1995, siendo la definición oficial de Ada 95 el documento denominado *Reference Manual for the Ada Programming Language* (ISO, 2000).

La Organización Internacional para la Estandarización (ISO) tiene una estructura jerárquica de varios niveles, dando lugar a un gran número de grupos de trabajo, especializados cada uno de ellos en un tema concreto. El grupo encargado del desarrollo de estándares ISO para el lenguaje de programación Ada es el ISO-IEC/JTC1/SC22/WG9 donde las siglas significan lo siguiente:

- JTC1: *Joint Technical Committee* 1, encargado de las estandarizaciones relativas a las tecnologías de la información.
- SC22: *Subcommittee* 22, trata los lenguajes de programación, sus entornos e interfaces con sistemas *software*.
- WG9: *Working Group* 9, tiene la responsabilidad de desarrollar los estándares ISO para el lenguaje Ada.

Dentro de este grupo, existen a su vez varios de los denominados *Rapporteur Groups* que tratan aspectos concretos del lenguaje, entre los que destaca la labor de los siguientes:

- Ada Rapporteur Group (ARG): discutiendo mejoras del lenguaje
- Annex H Rapporteur Group (HRG): asesorando en los aspectos de seguridad y fiabilidad de aplicaciones Ada.

A día de hoy ya está disponible una nueva revisión del lenguaje, Ada 2005. Debido al interés que tiene para esta tesis sus aspectos más relevantes se destacan en una sección aparte (ver 2.3.2). Tras cumplir con los últimos trámites administrativos dentro de la estructura ISO, la versión definitiva del manual de referencia puede consultarse en (Taft et al., 2007).

La definición de Ada tenía como meta dotar al nuevo lenguaje de una serie de características entre las que podemos destacar las siguientes: legibilidad, tipado fuerte, manejo de excepciones, abstracción, orientación a objetos y concurrencia. Ada es un lenguaje muy completo, mucho más extenso, por ejemplo, que Pascal o C, y quizás similar a C++. Está compuesto por un núcleo común, más una serie de anexos especializados, que no añaden sintaxis, pero sí una serie de nuevas bibliotecas y mecanismos de implementación que deben estar soportados si se quiere trabajar conforme a ellos.

Junto con el anexo de sistemas de tiempo real (anexo D, que tratamos a continuación), la gestión de la concurrencia es especialmente importante dentro de nuestro marco de trabajo pues, como ya hemos mencionado, en los sistemas de tiempo real es muy habitual que sucedan varias cosas en paralelo, siendo mucho más natural y cómodo manejarlo con herramientas integradas en el propio lenguaje que tener que recurrir a llamadas al sistema operativo.

El anexo de sistemas de tiempo real define una serie de características adicionales pensadas para aquellas implementaciones destinadas a formar parte de sistemas de tiempo real. De manera resumida, estas características son:

- Prioridad de la tarea: se utiliza la construcción *Pragma Priority*, o en su caso, *Pragma Interrupt_Priority* para asignarle una prioridad a una tarea, objeto protegido o subprograma.
- Planificación: permite que se describan las reglas que determinan qué tarea se ejecuta cuando hay varias disponibles para ello. La opción por defecto es *FIFO_Within_Priorities*, es decir, primero la que tiene la prioridad más alta y, en caso de igualdad, la primera en llegar.
- Protocolo de techo de prioridad: especifica el modo en que interaccionan la prioridad de la tarea y los techos de prioridad de los objetos protegidos. Por defecto se utiliza el protocolo *Ceiling_Locking* equivalente al techo de prioridad inmediato.
- Protocolos de encolado: permite especificar la manera en la que se ordena una cola de tareas que están esperando obtener un determinado servicio. El lenguaje define *FIFO_Queuing* y *Priority_Queuing*.
- Prioridades dinámicas: ofrece una serie de instrucciones que permiten cambiar la prioridad de una tarea en tiempo de ejecución.
- Tiempo: añade una biblioteca con facilidades para manejar un reloj monótono de alta resolución.

- Precisión de las instrucciones *delay*: Define con precisión los efectos de utilizar las instrucciones *delay* y *delay until* usadas para diferir trabajos.
- Transferencia síncrona de control: ofrece un tipo de semáforo que puede ser utilizado para suspender operaciones o implementar colas de alto nivel, todo ello de manera síncrona.
- Transferencia asíncrona de control: define una forma de suspender o reanudar tareas de manera asíncrona.
- Restricciones al modelo de tareas: detalla una serie de pragmas restrictivos que se pueden utilizar para prohibir la utilización de una serie de componentes del lenguaje, lo cual resulta muy útil cuando queremos construir sistemas especialmente eficientes o fácilmente analizables como veremos a continuación.

2.3.2. El nuevo estándar Ada 2005

De nuevo los constantes avances hicieron que se planteara la necesidad de revisar el estándar de Ada 95. Después de varios años de trabajo en diferentes mejoras, la nueva definición del lenguaje, Ada 2005, vio la luz en el mes de Marzo de 2007.

El nuevo estándar contiene una gran variedad de modificaciones sobre el lenguaje en general, pero el avance en todo lo relativo a programación de tiempo real y sistemas de alta integridad es especialmente significativo. Entre las nuevas herramientas podemos destacar las siguientes:

- Eventos temporizados: mecanismos que permiten al usuario ejecutar un procedimiento en un momento dado sin necesidad de recurrir a otras herramientas como tareas o instrucciones *delay*.
- Relojes de tiempo de ejecución: herramientas que permiten controlar en todo momento la cantidad de tiempo que una tarea ha ocupado el procesador.
- Temporizadores de tiempo de ejecución: temporizadores que se pueden programar a priori para que si una tarea consume más de un cierto tiempo de CPU se dispare una alarma.
- Cuotas de tiempo de ejecución para grupos de tareas o cuotas colectivas: semejantes a los temporizadores de tiempo de ejecución. La diferencia es que este nuevo

elemento está pensado para medir el tiempo de procesador consumido, no por una tarea, sino por un grupo de ellas. Este elemento lleva asociada la complejidad de decidir con qué algoritmo se recargan las cuotas.

- Procedimiento para finalizar una tarea: se añade un mecanismo para asignarle un procedimiento a una tarea, que será invocado en caso de que la tarea termine (generalmente debido a una instrucción *abort* o a una excepción no controlada). Así, se evita que una tarea muera de manera silenciosa, ya que esto podría poner el peligro la integridad del sistema.

- Inclusión del perfil de Ravenscar: hasta ahora, las características del perfil de Ravenscar (ver apartado 2.3.3) se incluían mediante un numeroso conjunto de pragmas. En Ada 2005 se añade un pragma específico con todas ellas, incluyendo las últimas modificaciones que se han realizado sobre el perfil.

- Planificación por bandas de prioridad: se permite definir una serie de bandas de prioridad con diferentes políticas de planificación, de manera que en un mismo sistema pueden convivir varios grupos de tareas planificados cada uno de ellos de la forma más eficiente.

- Soporte para planificación EDF y Round-Robin: Se incluye en el lenguaje la política de planificación EDF (ver sección 2.2.3) y Round-Robin.

- Planificación cooperativa (sin desalojo): se añade una política de planificación sin desalojo, de manera que cuando una tarea accede al procesador, lo ocupa hasta que termina su trabajo. Este tipo de planificación facilita la implementación de ejecutivos cíclicos (ver sección 2.2.1).

- Techos de prioridad dinámicos: hasta ahora el techo de prioridad de los objetos protegidos era estático. Esta nueva medida permite que se modifiquen dinámicamente, lo cual puede ser muy útil a la hora de construir sistemas con varios modos de funcionamiento o sistemas con prioridades dinámicas en general.

Veremos más adelante que estos cambios en el estándar Ada son de vital importancia para la arquitectura desarrollada en esta tesis, ya que gran parte de la innovación que se propone debe ser implementada con la ayuda de estos nuevos elementos.

2.3.3. El perfil de Ravenscar

Como ya se ha mencionado, el lenguaje Ada es muy potente, dispone de un buen número de elementos que permiten una gran variedad de construcciones. Esto, que es positivo para construir sistemas de propósito general, se vuelve en nuestra contra a la hora de construir sistemas de alta integridad ya que, cuanta más variedad de construcciones haya, mayor será el indeterminismo asociado. De hecho, a día de hoy no es posible analizar el comportamiento temporal de una aplicación que haga uso de todas las facilidades que ofrece Ada.

El IRTAW (*International Real-Time Applications Workshop*) es una reunión de trabajo de carácter científico que tiene en los sistemas de tiempo real su foco de atención, haciendo especial énfasis en el lenguaje Ada. Una de las aportaciones más importantes realizadas en este taller es el denominado *Perfil de Ravenscar* (Burns, 1999) (llamado así porque la reunión en la que se definió se celebró en ese lugar del Reino Unido). El perfil de Ravenscar fue refinado en las dos siguientes ediciones (celebradas en 1999 y 2001). Posteriormente, se han hecho algunas modificaciones para adaptarlo a Ada 2005 (de la Puente and Zamorano, 2003), siendo la más importante de ellas que el perfil pasa a ser parte del propio lenguaje gracias una directiva del compilador (*pragma*) que permite a una aplicación solicitar el modo de operación restringido característico del perfil de Ravenscar.

La estrategia seguida para definir el perfil puede resumirse en tres puntos:

- Eliminar todos aquellos componentes que introducen indeterminismo o bien suponen una carga muy pesada en tiempo de ejecución.
- Mantener aquellas utilidades que permiten construir sistemas de tiempo real y alta criticidad que satisfagan las necesidades actuales.
- Asegurarse de que es posible realizar un análisis de tiempo de respuesta, así como otras clases de análisis estáticos en los programas construidos conforme al perfil.

Para comprender mejor el perfil vamos profundizar en el detalle de cuáles son las principales acciones que se permiten o prohíben para lograr los objetivos marcados.

El modelo de tareas se restringe a las siguientes situaciones:

- El conjunto de tareas es estático (no hay creación dinámica de tareas).

- Las tareas se ejecutan cíclicamente y sin fin. Tienen un solo punto de suspensión y contienen código que se ejecuta secuencialmente sin llamadas bloqueantes.
- Las tareas pueden ser periódicas (con periodo conocido) o esporádicas, en cuyo caso se debe conocer el tiempo mínimo entre dos activaciones consecutivas. El tiempo de cómputo en el peor caso posible (WCET) también debe ser conocido, así como el plazo de respuesta *deadline* para poder llevar a cabo el análisis estático.
- La comunicación entre tareas se realiza mediante variables compartidas (objetos protegidos sin *entry*), salvo en el caso de tareas esporádicas, cuya activación se realiza mediante un objeto protegido con una sola *entry*.
- La planificación se realiza mediante prioridades fijas y esquema FIFO para las tareas con el mismo nivel de prioridad. También se pueden analizar esquemas no desalojables.
- El protocolo de acceso a objetos protegidos es el de techo de prioridad inmediato (consultar 2.2.4).

Se eliminan los elementos que producen indeterminismo en el uso de la memoria:

- Creación dinámica de atributos de tareas.
- Asignación implícita de espacio de memoria en la pila estándar.

Se eliminan los elementos que producen indeterminismo temporal:

- Sólo puede haber una tarea esperando en un *entry*.
- La barrera de un objeto protegido debe ser sencilla (una expresión *booleana*).
- El reencolado está prohibido, así como la asignación dinámica de manejadores de interrupción y la transferencia asíncrona de control.

Además, se utiliza una base temporal precisa mediante el uso del reloj de tiempo real *Ada.Real_Time* o similar. No se permite el uso del paquete *Ada.Calendar* ni retardos relativos (sólo se permite el uso de la instrucción *delay until*).

Gracias a las restricciones mencionadas es posible crear un núcleo de tiempo real, con soporte para concurrencia (restringida), suficientemente pequeño y sencillo como para

que pueda ser certificado (ver sección 3.1), y al mismo tiempo eficiente y analizable, lo cual lo hace muy útil para dar soporte a sistemas de alta integridad (Burns et al., 2003; Vardanega and Caspersen, 2001), garantizando además que una aplicación diseñada conforme a Ravenscar puede ser analizada (Zamorano and de la Puente, 2002; Vardanega et al., 2005). El apéndice A enumera todas las restricciones que impone el perfil tanto en lenguaje natural como mediante las directivas que hay que darle al compilador para garantizar su cumplimiento por parte de la aplicación.

2.3.4. ORK: un núcleo de tiempo real

Existen varias implementaciones de núcleos de tiempo real que soportan el perfil de Ravenscar, algunas de ellas comerciales, tales como ObjectAda Real-Time Raven (fabricado por Aonix) (Burns et al., 1998) o GSTART (fabricado por Green Hill Software) (GST, 2005), y otras libres como ORK (*Open Ravenscar real-time Kernel*), desarrollado en el grupo de sistemas de tiempo real de la Universidad Politécnica de Madrid (de la Puente, Ruiz and Zamorano, 2000; de la Puente, Ruiz, Zamorano, García and Fernández-Marina, 2000; de la Puente et al., 2001; Zamorano and Ruiz, 2003).

Para desarrollar parte del trabajo ha sido necesario utilizar uno de estos núcleos como punto de partida. En nuestro caso particular hemos optado por ORK ya que, además de ser *software* libre, contamos con la ventaja del conocimiento acumulado sobre la herramienta tras varios años de continuo desarrollo y mantenimiento. De hecho, ya se han publicado varios trabajos sobre ORK relacionados con alguno de los temas que se exponen en esta tesis (Zamorano et al., 2004; Urueña et al., 2005; Pulido et al., 2005).

ORK es un núcleo mínimo de altas prestaciones que proporciona soporte para ejecutar programas acordes con el perfil de Ravenscar de Ada y también para el lenguaje C. Existen dos versiones de ORK, según se quiera ejecutar el código resultante en un PC desnudo o en una plataforma ERC32. Esta última plataforma, protegida frente a radiación, es muy utilizada en el entorno aeroespacial, especialmente por la Agencia Espacial Europea, con la que colaboramos activamente a través de diversos proyectos, por lo que nuestro trabajo partirá de la versión de ORK para ERC32.

ORK está integrado con el compilador GNAT según se muestra en la figura 2.8, de manera que ORK proporciona el soporte básico para gestionar a bajo nivel las hebras, el tiempo, la memoria, las interrupciones, y la salida serie, mientras que por encima, la capa GNARL (*GNU Ada Runtime Library*) sirve de interfaz para dar soporte al modelo de tareas de Ada. En realidad, esta capa está dividida en dos, ya que a la parte de más bajo

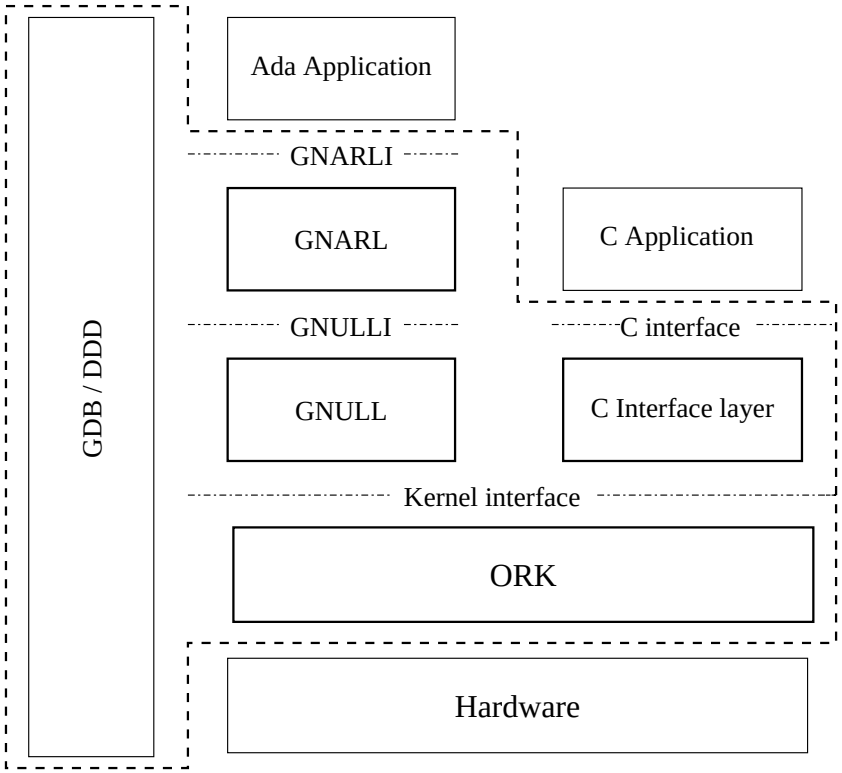

Figura 2.8: Arquitectura del sistema GNAT-ORK

nivel, que es dependiente del *hardware* y del sistema operativo se la denomina GNULL.

2.4. El proceso de desarrollo en ASSERT

Otro de los flancos que se quiere abordar en esta tesis está situado en un nivel de abstracción más alto. Actualmente, se calcula que los costes de las fases de integración y pruebas de un proyecto pueden llegar a ascender hasta el 50 % de los costes totales. Generalmente, esto es debido a una cadena de errores durante el proceso de elaboración del producto, especialmente en las etapas tempranas del mismo: unas especificaciones poco claras o ambiguas, errores en la implementación, etc. Para minimizar este problema se propone utilizar lenguajes de modelado no ambiguos, que permiten al diseñador del sistema situarse en un nivel de abstracción más alto, obviando detalles de la implementación, para que posteriormente una herramienta se encargue de generar código fuente de manera automática, de esta forma se eliminan los errores introducidos por el factor humano. Den-

tro del proyecto ASSERT, en el que hemos colaborado activamente durante los últimos 3 años, tienen especial relevancia dos lenguajes: AADL y UML, de los que comentamos los aspectos más significativos en los siguientes apartados.

2.4.1. AADL

AADL son las siglas en inglés de *Lenguaje de Análisis y Diseño de Arquitecturas* (SAE, 2004). Es un lenguaje definido por la SAE (*Society of Automotive Engineers*), con la colaboración de diferentes industrias e instituciones del entorno aeroespacial, tales como Airbus, la Agencia Espacial Europea, Boeing, EADS, etcétera. Está pensado para sistemas de tiempo real de alta integridad, y se utiliza para describir la estructura de sistemas como un conjunto de componentes *software* que se ejecutan en una plataforma *hardware*. El lenguaje incluye elementos que permiten expresar las interfaces funcionales entre componentes y su interacción, por ejemplo, mediante puertos de entrada y salida de datos, así como aspectos críticos relacionados con la planificación o el comportamiento dinámico al dar soporte a cambios de modo.

AADL es un lenguaje donde el elemento básico es el componente. Se puede establecer una jerarquía de componentes y diversas relaciones entre ellos tal y como veremos a continuación. En primer lugar se define un *tipo* componente, que especifica su interfaz con el exterior, declarando sus funcionalidades y los subcomponentes que son accesibles desde el exterior. Más tarde, se define una *implementación* del componente, en la que se refina la especificación y se añaden las partes privadas del componente y los modos de funcionamiento que permiten configurarlo de diversas maneras.

Los mayoría de los componentes están clasificados en dos grandes grupos: componentes de plataforma de ejecución y componentes *software*. En el primer grupo se encuentran:

- Procesador: modela una abstracción de *hardware* y *software* capaz de planificar y ejecutar hebras.
- Memoria: representa dispositivos de almacenamiento, tales como RAM, ROM, discos o cintas.
- Bus: componente a cargo de trasladar información de un componente a otro.
- Dispositivo: elemento *hardware* cuyo funcionamiento interno es irrelevante para el resto del sistema. Sólo es importante su interfaz para interactuar con el resto del sistema. Ejemplos típicos serían sensores, actuadores o displays.

Entre el grupo de componentes *software* podemos encontrar los siguientes:

- Dato: representa un tipo dato de código fuente, que puede ser compartido. Se supone que cuenta con algún mecanismo para controlar la concurrencia y puede tener sus propias operaciones.
- Subprograma: representa una clásica función o procedimiento de código fuente.
- Hebra: flujo de control secuencial planificable que ejecuta instrucciones dentro del espacio de memoria virtual de un proceso.
- Grupo de hebras: grupo lógico de hebras. Simplemente se utiliza para simplificar el diseño cuando se trabaja con sistemas de gran tamaño.
- Proceso: modela un espacio virtual de memoria, cuya implementación contiene al menos una hebra o un grupo de ellas.

Además de los grupos anteriores, en AADL se pueden definir los siguientes elementos:

- Sistema: componente que representa el mayor nivel de abstracción, contiene tanto la plataforma de ejecución como el *software* que corre sobre ella, además de otros sistemas como subcomponentes, si resulta necesario.
- Paquete: no es un componente en sí, sino una manera de organizarlos en grupos de declaraciones con espacios de nombres diferentes.

En AADL se establecen relaciones entre los distintos componentes mediante características, conexiones y flujos. Las características se utilizan para especificar el interfaz que presenta un componente al exterior. Hay cuatro categorías:

- Puerto: se utiliza para representar la transmisión asíncrona de datos y/o control, pudiendo declararse como de entrada, salida y de entrada-salida.
- Subprograma: su definición es idéntica al componente *software* con el mismo nombre.
- Parámetro: representa los datos intercambiados por un subprograma.
- Acceso a subcomponente: simboliza un acceso tanto de entrada como de salida a datos o buses internos.

Una conexión, como su nombre indica, representa una transferencia de datos y/o control entre dos componentes. Podemos hablar de conexiones entre puertos, entre parámetros y entre accesos a subcomponentes.

Un flujo es una corriente lógica de información a través de una serie de componentes y conexiones que facilita la capacidad de soportar diferentes tipos de análisis, como por ejemplo, la latencia de principio a fin de cierta comunicación.

Por último, otro elemento fundamental de AADL son las propiedades. Se utilizan para proveer un amplio rango de información acerca de componentes, flujos, conexiones, etcétera. Las propiedades AADL tienen nombre, tipo y valor. Hay dos grupos de propiedades predefinidas por el lenguaje, que son las utilizadas más habitualmente, pero se permite que el usuario especifique todas las que considere necesario para realizar su trabajo. Esto hace que AADL sea un lenguaje abierto que puede llegar a cubrir la gran mayoría de las necesidades del diseñador del sistema.

2.4.2. HRT-UML/RCM

El Lenguaje Unificado de Modelado (UML) (Booch et al., 1998), y sobre todo su nueva versión UML2 (Obj, 2006; Booch et al., 2005), tiene actualmente una gran relevancia tanto en la comunidad científica como en la empresarial, pues es la notación más utilizada a la hora de especificar y documentar sistemas a lo largo de todas sus fases.

Sin embargo, el lenguaje UML, al ser de propósito general, resulta demasiado genérico a la hora de tratar aspectos determinados. Para poder tratar con detalle conceptos pertenecientes a ciertos dominios se recurre a los *perfiles* (Fuentes and Vallecillo, 2004), que son una herramienta del lenguaje que permite extender tanto su semántica como su análisis con objeto de representar esas características especiales. Algunos ejemplos de perfiles UML son:

- SPT: *UML Profile for Schedulability, Performance and Time* para utilizar en el dominio del tiempo, la planificación y temas relacionados con el funcionamiento del sistema.
- Qos-FT: *UML Profile for Quality of Service and Fault Tolerance*, especializado en conceptos de calidad de servicio y tolerancia a fallos.
- CCM: *UML Profile for CORBA Component Model*, para aplicaciones basadas en CCM.

Otro de estos perfiles ha sido elaborado recientemente para tratar las características de los sistemas basados en Ravenscar y se denomina HRT-UML/RCM. Este perfil es el eslabón actual de una larga cadena de lenguajes que se han utilizado durante los últimos veinte años para modelar sistemas de tiempo real. RCM son las siglas de *Ravenscar Computational Model*, y es el sucesor de HRT-UML (Mazzini et al., 2003), que vino después de HRT-HOOD (Burns and Wellings, 1994; Burns and Wellings, 1995), que a su vez proviene de HOOD (Robinson, 1992). El perfil RCM se ha construido a partir de los siguientes conceptos:

- Entidad esporádica: aquella que contiene una hebra que se activa mediante un evento externo, ya sea *software* o *hardware*.
- Entidad cíclica: análoga a la anterior, pero se activa periódicamente debido al paso natural del tiempo.
- Entidad protegida: recurso del sistema que sigue algún protocolo de acceso a medios compartidos para garantizar la exclusión mutua.
- Entidad pasiva: noción ligada al concepto de agregación, una entidad pasiva en un recurso del sistema que no tiene ningún protocolo de acceso.
- Entidad activa: de manera intuitiva, una entidad activa es aquella que se sitúa fuera del modelo, contiene hebras y su comportamiento no interfiere con él.

Tanto AADL como el perfil UML para Ravenscar serán utilizados por otros grupos de investigación para describir los contenedores mencionados en uno de los objetivos de esta tesis. Dichos contenedores son los elementos que permitirán facilitar el diseño del sistema y generar automáticamente código fuente ejecutable sobre la plataforma que proponemos más adelante, avanzando así en el diseño de sistemas de alta integridad.

Capítulo 3

SISTEMAS PARTICIONADOS

3.1. Certificación de sistemas críticos

Como ya se ha comentado en capítulos anteriores, un error en un sistema crítico puede tener consecuencias catastróficas, por ello, es necesario asegurarse de que el producto final cumple con las garantías necesarias. La manera más habitual de comprobar la integridad de un producto es recurrir a un proceso de certificación.

Éste es un proceso llevado a cabo por una entidad reconocida e independiente de las partes interesadas que manifiesta la conformidad, en este caso del sistema, con las especificaciones dadas, así como con las definidas en alguna norma específica del dominio de la aplicación, ya que en la mayoría de los sectores industriales podemos encontrar algún estándar de certificación, tales como: DO-178B utilizado en sistemas de aviación civil, IEC 601-4 en equipamiento médico, IEC 880 en plantas de energía nuclear, EN 50128 en los ferrocarriles europeos, etcétera.

Puesto que el radio de acción principal del presente trabajo son las aplicaciones aeroespaciales, comentamos algunos detalles del estándar DO-178B, titulado "*Software Considerations in Airborne Systems and Equipment Certification*", por el que deben regirse los productos de este área.

El estándar DO-178B fue creado por la RTCA (*Radio Technical Commission for Aeronautics*) y actualmente está aceptado como el instrumento para certificar el *software* de aviación nuevo. Está enfocado principalmente en los procesos de desarrollo y como consecuencia, obtener una certificación DO-178B requiere entregar una gran cantidad de

documentos y registros relacionados con las fases de desarrollo del sistema.

No toda la certificación se realiza al mismo nivel. El estándar DO-178B establece cinco niveles de certificación, de manera que cada uno de ellos se corresponde con la severidad de las consecuencias que un fallo en potencia pudiere ocasionar. Los cinco niveles de criticidad son los siguientes:

- Nivel A ⇒ Consecuencias catastróficas
- Nivel B ⇒ Consecuencias graves
- Nivel C ⇒ Consecuencias serias
- Nivel D ⇒ Consecuencias leves
- Nivel E ⇒ Sin consecuencias

El estándar define cerca de una veintena de documentos y registros con diferente información (Ej: *Software Development Plan* (SDP), *Software Verification Plan* (SVP), *Problem Reports*, etc.). Dependiendo del nivel al que se pretenda certificar una aplicación se siguen unos procedimientos u otros, es decir, el rigor del control de calidad y la documentación requerida varían en función del nivel de certificación deseado.

Certificar un producto al máximo nivel es muy difícil y caro, ya que sólo es posible llegar a cumplir con las exigencias del *"nivel A"* en sistemas muy poco complejos y después de muchas horas de trabajo y preparación. Por este motivo, es muy importante a la hora de diseñar un sistema fijar desde el inicio cuál es el nivel de certificación más adecuado para cada parte. Por ejemplo, no es extraño que un proyecto espacial contenga aplicaciones certificadas a 3 niveles diferentes.

Como ya adelantábamos en la introducción, uno de nuestros objetivos es proporcionar un método que permita ejecutar varias aplicaciones en un mismo procesador de manera segura y eficiente. Sin embargo, al estar las aplicaciones certificadas a distintos niveles podemos asegurar que el control de calidad durante su desarrollo ha sido diferente, y por tanto también su nivel de fiabilidad es distinto. Por ello, no se deben ejecutar aplicaciones de distinto nivel de criticidad en una misma plataforma sin proporcionar antes una serie de medidas de seguridad extra, ya que un fallo en una aplicación de baja criticidad (y por ende poco fiable al haber superado unos controles menos estrictos), podría provocar un efecto colateral y comprometer la integridad de las partes vitales del sistema a pesar de que estas últimas no contuvieran errores internos.

3.2. Aislamiento y planificación jerárquica

Visto el proceso de certificación, queda claro que para integrar varias aplicaciones de distinto nivel de criticidad en una misma plataforma de ejecución es necesario incorporar medidas de aislamiento y contención de fallos que eviten que un error interno a una aplicación se extienda y afecte a las demás. Es decir, hay que habilitar unas *particiones virtuales* que garanticen que cada aplicación se ejecute de manera segura y aislada, sin injerencias de las demás. Esto implica que se debe proporcionar:

- Aislamiento temporal: el tiempo de ejecución de una tarea en el peor caso posible se supone conocido en tiempo de diseño, sin embargo, en tiempo de ejecución, esta estimación puede ser sobrepasada por diversos motivos (cálculos incorrectos, activaciones de tareas esporádicas fuera del modelo previsto, bucles infinitos, etc.), por lo que una tarea puede intentar ocupar el procesador durante demasiado tiempo. Por este motivo, es necesario disponer de alguna herramienta que evite que una tarea monopolice el uso del procesador, impidiendo que otras tareas puedan cumplir sus plazos previstos.
- Aislamiento espacial: de manera análoga al tiempo de ejecución, las aplicaciones también disponen de un espacio de memoria que no debe ser invadido por otras aplicaciones. Sin embargo, un error de programación podría provocar el desbordamiento de una pila, lo cual, con toda probabilidad dejaría inutilizables las aplicaciones contiguas. Por tanto, es igualmente necesario adoptar medidas que garanticen que no se producirán desbordamientos de las zonas de memoria asignadas a cada aplicación.

Una técnica muy apropiada para proporcionar aislamiento temporal entre aplicaciones es la *planificación jerárquica* (Mok et al., 2001; Regehr et al., 2003). Conceptualmente es una idea sencilla aunque tras la noción general se esconden múltiples variaciones, cada una de ellas con sus ventajas y sus inconvenientes.

Dado un sistema con varias aplicaciones, donde cada una de ellas contiene a su vez un conjunto de tareas, el argumento principal de la planificación jerárquica consiste en dividir el sistema en varias particiones, que típicamente coincidirán con las aplicaciones, y utilizar dos clases de planificadores en la plataforma en que se ejecutan. De esta manera se distingue entre:

- Planificador global: único en el sistema, se encarga de gestionar particiones. Siguiendo las normas dictadas por una política de planificación (las repasadas en las

sección 2.2 u otras), es el responsable de decidir cuál es la partición que debe ejecutarse en cada momento, la cual recibe el nombre de *partición activa*.

- Planificadores locales: cada partición tiene asignado un planificador local, de manera que cuando el planificador global activa un partición, es el planificador local asignado a esa partición el encargado de decidir entre todas las tareas que hay en ella cuál es la que debe ocupar el procesador. Un dato importante a tener en cuenta es que cada planificador local podría regirse según una política de planificación diferente, dotando al sistema de mayor flexibilidad.

Vemos, por tanto, que hay múltiples opciones para planificar un sistema de manera jerárquica, utilizando planificación estática, prioridades fijas, prioridades dinámicas, etc. Las siguientes secciones están dedicados a analizar las opciones más importantes debido a su grado de utilización o a sus aportaciones novedosas.

3.3. Particionado estático

El particionado estático está basado en una teoría similar a la descrita en la sección 2.2.1. La idea consiste en que el tiempo se divide en ranuras a las que se asignan particiones estáticamente. Seguidamente, cuando una partición se activa, es el planificador local el encargado de gestionar esa ventana de tiempo despachando convenientemente las tareas locales.

3.3.1. Particionado y comunicaciones en ARINC 653

Uno de los ejemplo más conocidos que siguen este enfoque es el estándar ARINC 653 (*Aeronautical Radio, Inc*) (ARI, 2003), que forma parte de IMA (*Integrated Modular Avionics*, término utilizado para describir un sistema de computación distribuido y de tiempo real ampliamente utilizado en aeronaves).

En ARINC 653 se modela un ciclo principal estático que será el que se repita continuamente. En este ciclo principal se establece qué partición se ejecutará en cada ranura así como la longitud de éstas, asignando al menos una ranura a cada partición. El planificador global se encarga por tanto de activar las diferentes particiones siguiendo el esquema descrito, tal y como muestra la figura 3.1, activando en cada momento la partición adecuada. Una vez que se agota el plazo asignado la partición es desalojada y pasa a

activarse la siguiente. No hay prioridades a nivel global. Cuando una partición está activa, su planificador local se encarga de decidir la tarea que debe ocupar el procesador, esta vez sí, siguiendo un esquema de prioridades fijas.

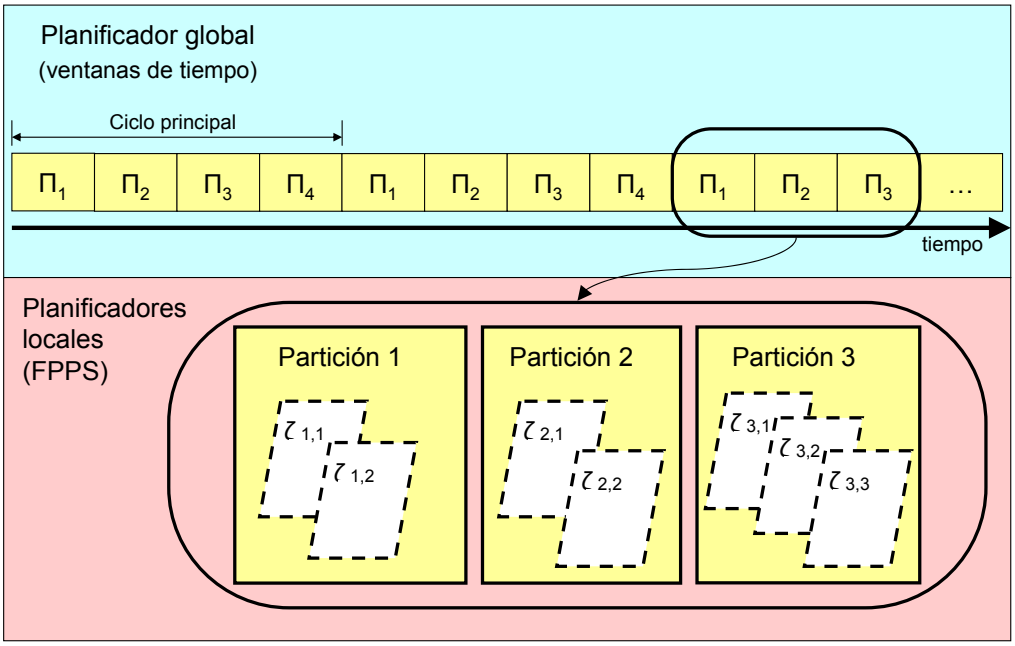

Figura 3.1: Planificación siguiendo el estándar ARINC 653

3.3.2. Análisis de planificación en ARINC 653

Una vez descritas las características más importantes de la arquitectura ARINC 653, a continuación se detalla el análisis de tiempo de respuesta que debe realizarse en un sistema construido conforme a esta arquitectura. Seguidamente, incluimos un resumen de los pasos que se deben seguir (puede consultarse el desarrollo completo en el trabajo (Audsley and Wellings, 1996)).

El modelo de particionado establecido en ARINC 653 permite separar el análisis de las diferentes particiones Π_i, uniendo la contribución al tiempo de respuesta de todas las demás particiones del sistema en una sola tarea τ_0 (ver figura 3.2), con los parámetros que se detallan a continuación:

$$C_0 = T_i^{\Pi}; T_0 = T_i^{\Pi}; O_0 = C_i^{\Pi}$$

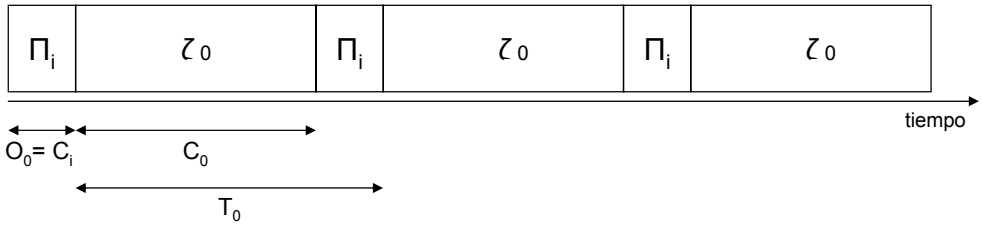

Figura 3.2: Parámetros para realizar el análisis en ARINC 653

La condición de viabilidad para el sistema completo viene dada por la siguiente expresión:

$$\forall \Pi_j \in \Pi; \forall \tau_i \in \Pi_j; R_i + J_i \leq D_i$$

en la que R_i es el tiempo de respuesta de la tarea τ_i dado por la ecuación recursiva 3.1:

$$R_i = C_i + B_i + L_i + OS_i(0, R_i) + \text{máx}\left(0, \left\lceil \frac{R_i - O_0}{T_0} \right\rceil \cdot C_0\right) + \sum_{\tau_j \in hp(\tau_i)} \left\lceil \frac{R_i}{T_j} \right\rceil \cdot C_j \quad (3.1)$$

donde:

- J_i es la fluctuación (*jitter*) de la tarea τ_i.
- C_i representa el tiempo de ejecución en el caso peor de la tarea τ_i.
- B_i cuantifica el tiempo que la tarea τ_i permanece bloqueada por tareas de menor prioridad (dentro de la misma partición) debido a recursos compartidos.
- L_i modela el plazo que la ejecución de una partición puede ser diferida para que la anterior salga de una región crítica.
- $OS_i(t_1, t_2)$ representa la máxima sobrecarga en el intervalo $[t_1, t_2)$ provocada por las capas inferiores del sistema operativo.
- O_0, T_0 y C_0 representan la fase (*offset*), periodo y tiempo de cómputo de la tarea acumulada τ_0 (ver figura 3.2)
- $hp(\tau_i)$ es el conjunto de tareas que existen en la misma partición que la tarea bajo análisis pero que tienen un nivel de prioridad más alto.

3.3.3. Valoración

El modelo ARINC 653 tiene como ventaja su gran determinismo, ya que la planificación de las particiones está predeterminada por el plan que se repite periódicamente. Sin embargo, comparte los mismos inconvenientes que los métodos de planificación estática en general, ya que es un modelo muy rígido, en el que resulta muy difícil hacer cambios en la planificación de las particiones. Así pues, podemos destacar los siguientes problemas:

- Cambiar la configuración del sistema, ya sea añadiendo o eliminando particiones no es trivial.
- El tiempo asignado a cada ranura es fijo, lo cual no da opción a reutilizar el tiempo desperdiciado por una partición.
- Incluir tareas esporádicas en este esquema no es ni sencillo ni eficiente.

Merece una mención especial el esquema de comunicaciones exclusivamente basado en mensajes empleado en ARINC 653 (impuesto por IMA), que consiste en que una capa de transporte pone a disposición de las particiones una ventana de transmisión periódica y de duración predeterminada durante la cual las aplicaciones pueden enviar sus mensajes sin riesgo de colisiones, ya que en cada ventana sólo transmite una partición.

Este esquema tan rígido como el de asignación de ranuras de tiempo de procesador a particiones también tiene el inconveniente de que construir la agenda global no es nada sencillo, especialmente en sistemas con un alto nivel de distribución y muchas aplicaciones independientes. Además, no debe despreciarse el hecho de que cuando un mensaje se envía no podrá ser leído hasta que la partición de destino se active, independientemente de la urgencia del mensaje.

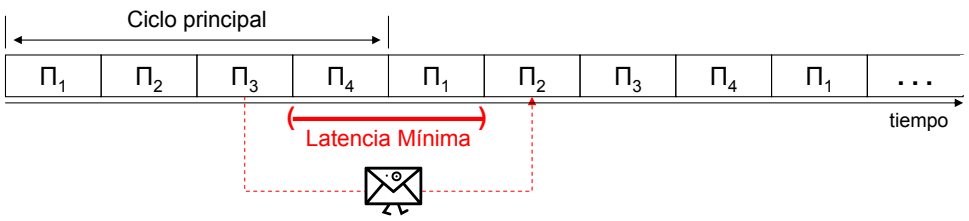

Figura 3.3: Paso de mensajes en ARINC 653

La figura 3.3 muestra una de estas situaciones, en la que la partición Π_3 envía un mensaje que debe ser entregado a la partición Π_2. Puede verse fácilmente cómo desde que

el mensaje es enviado hasta que puede ser leído por la aplicación hay una latencia mínima debido a las ranuras que ocupan otras particiones, por tanto, cada vez que se produzca una comunicación de estas características el mensaje estará en espera una cantidad de tiempo no despreciable, independientemente de si es importante para la integridad del sistema e incluso aunque las particiones intermedias desperdicien totalmente su tiempo de ejecución porque no tengan tareas pendientes de completar.

Actualmente ARINC 653 es el modelo más utilizado a bordo de aeronaves, ya que hasta ahora ha primado la seguridad ofrecida por esta arquitectura frente a las desventajas que hemos destacado. Podemos encontrar trabajos basados en implementar esta arquitectura usando el lenguaje Ada (Dobbing, 2000; Tokar, 2003) y, de hecho, existe un sistema operativo comercializado por *Green Hills* certificado al máximo nivel de criticidad (nivel A), que soporta Ada y está siendo usado en un amplio rango de proyectos por numerosas empresas pertenecientes al sector de la aviación, como son: Lockheed Martin, Boeing, Rockwell, Raytheon, BAE Systems, BF Goodrich Aerospace o Alenia Aerospatiale (Int, 2005).

Por otra parte, un buen número de empresas e instituciones, tales como EADS, Alcatel, la Agencia Espacial Europea y algunas de las ya citadas en el párrafo anterior están involucradas en proyectos como ASSERT con la intención de explorar la posibilidad de construir sistemas más flexibles que ARINC, de los que se pueda obtener un mayor rendimiento (abaratando por tanto los costes), manteniendo el nivel de seguridad exigido.

3.4. Particionado basado en servidores

3.4.1. Fundamentos de los servidores

Los servidores fueron presentados en la década de los 80 como una solución al problema de planificar tareas esporádicas dentro de un esquema de prioridades fijas como RMS (Sha et al., 1986). A pesar de RMS se había presentado como un algoritmo óptimo, su uso no estaba extendido debido a diversas dificultades prácticas como la mencionada. Mediante servidores el problema se ataca reservando un cierto ancho de banda para procesar los eventos esporádicos, de manera que estos son almacenados hasta que el servidor se hace con el procesador utilizando dicha reserva y procede a atenderlos.

Esta idea es fácilmente adaptable a un esquema de planificación jerárquica, que además avanza en la dirección indicada en la sección anterior, eliminando parte de la rigidez im-

puesta por el esquema cíclico del estándar ARINC 653.

Partiendo de los trabajos iniciales realizados en la planificación jerárquica con servidores (Deng et al., 1996), que imponían un gran número de restricciones al modelo computacional para ser capaces de realizar el análisis de planificación, poco a poco se ha ido desarrollando una densa teoría que ha permitido ir eliminando buena parte de esas restricciones, así como también se ha propuesto un gran número de variantes en busca de una mayor eficiencia de la arquitectura (Deng and Liu, 1997; Kuo and Li, 1998; Saewong et al., 2002; Almeida and Pedreiras, 2004; Lipari and Bini, 2005).

El componente básico de la arquitectura es claramente el *servidor*, que en este caso podría ser definido como un elemento especializado en la planificación y ejecución de tareas. El funcionamiento es sencillo: cada servidor tiene una cuota de tiempo de ejecución, de manera que cuando uno de ellos es activado por el planificador global, comienza a ejecutar tareas y, por tanto, a consumir esa cuota hasta que, o bien se agota, o bien es desalojado del procesador por el planificador global. Una vez agotada su cuota, el servidor permanece inactivo hasta que es recargada.

En otras palabras, el sistema está compuesto por un planificador global encargado de activar en cada momento el servidor apropiado. Los servidores a su vez, desempeñan la función de planificadores locales, despachando tareas cuando son activados por el planificador global. De esta manera, por cada partición que se desea realizar en el sistema se debe utilizar un servidor, que será el encargado de garantizar el ancho de banda definido por el diseñador.

Uno de los puntos claves de la arquitectura basada en servidores viene dada por los grados de libertad que tiene el diseñador del sistema, ya que existe la posibilidad de configurar el planificador global con diferentes políticas de planificación tales como FPPS (Fixed Priority Preemptive Scheduling), EDF, etc. Además, veremos más adelante que una vez dentro de una partición también es posible configurar cada servidor con múltiples opciones que aumentan aún más dicha flexibilidad.

Un ejemplo de planificación con servidores es la arquitectura propuesta en el proyecto FIRST (Davis and Burns, 2005a; Davis and Burns, 2005b), (ver figura 3.4). Aquí, el planificador global se rige por una política de prioridades fijas con desalojo (FPPS), así que es necesario asignarle a cada servidor un nivel de prioridad para que en cada momento el planificador global pueda asignarle el procesador al servidor con la prioridad más alta. Además se han definido tres tipos de servidores, que se distinguen por la manera que tienen de activarse y consumir su cuota de tiempo de ejecución. Éstos son:

- Servidor periódico (Sha et al., 1986): se activa con un periodo fijo y, siempre y cuando no sea desalojado, consume su cuota de tiempo de ejecución tanto si tiene tareas listas para ejecutar como si no las tiene. La cuota de tiempo de ejecución se repone en el comienzo del siguiente periodo.
- Servidor aplazable (Strosnider et al., 1995): similar al anterior, la diferencia es que, en este caso, si no hay ninguna tarea lista para ser ejecutada el servidor suspende su ejecución con objeto de preservar su cuota de tiempo por si algunas de sus tareas se activa en la parte final de su periodo. La cuota de tiempo de ejecución de este servidor se repone al comienzo del siguiente periodo tanto si ha sido consumida como si no lo ha sido.
- Servidor esporádico (Sprunt et al., 1989): en este caso cuando consume parte de su cuota, se programa la reposición de esa cantidad después de un cierto periodo T_s que varía en función de la planificación interna de sus tareas.

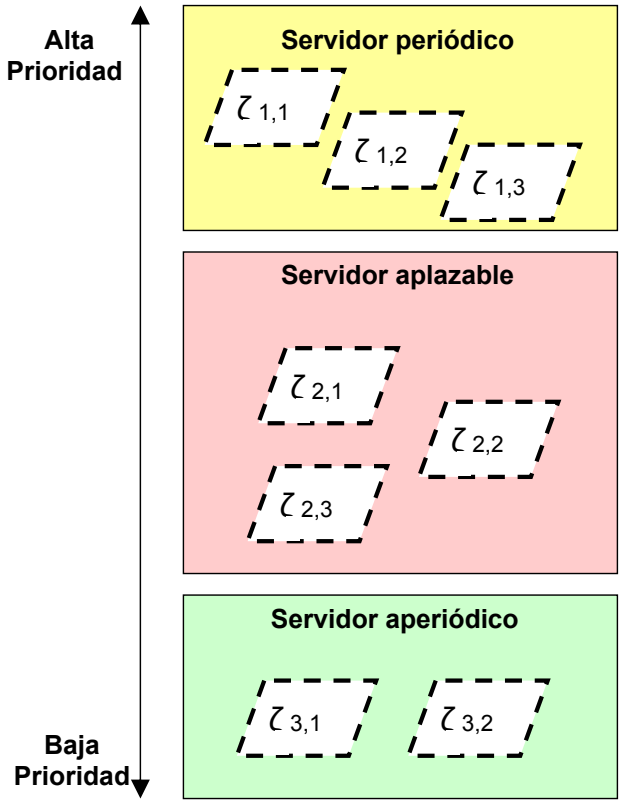

Figura 3.4: Arquitectura basada en servidores

3.4.2. Análisis de planificación con servidores

Dada la gran variedad de parámetros configurables que ofrece esta arquitectura, no es posible condensar en un sólo apartado todas las variantes de análisis que se pueden ofrecer, por lo que en este caso se recomienda recurrir a la bibliografía especializada para profundizar en el tema (Davis and Burns, 2005a; Davis and Burns, 2006). A modo de ejemplo, consideramos el caso general, en el que para hallar el tiempo de respuesta de una tarea se utiliza la ecuación recursiva 3.2:

$$w_i^n = L_i(w_i^n) + \left\lceil \frac{L_i(w_i^n)}{C_s} \right\rceil (T_s - C_s) + \sum_{\forall X \in hp(s)} \left\lceil \frac{\text{máx}\left(0, w_i^n - \left(\left\lceil \frac{L_i(w_i^n)}{C_s} \right\rceil T_s - C_s\right)\right) + J_X}{T_X} \right\rceil C_X \tag{3.2}$$

en la que se puede iniciar la recurrencia con un valor de $w_i^0 = C_i + \left\lceil \frac{C_i}{C_s} \right\rceil (T_s - C_s)$. Además, el valor de la expresión $L_i(W_i^n)$ debe ser actualizado en cada iteración utilizando la ecuación 3.3

$$L_i(w) = C_i + \sum_{\forall j \in hp(i)} \left\lceil \frac{w}{T_j} \right\rceil C_j \tag{3.3}$$

donde:

- C_s, T_s y J_s representan respectivamente la capacidad, el periodo de relleno y el *jitter* del servidor con prioridad *s*.
- C_X, T_X y J_X son los valores correspondientes al servidor de mayor prioridad.
- $hp(s)$ representa el conjunto de servidores con prioridad mayor que el servidor *s*.
- $L(w)$ es la carga debido a tareas activadas en un intervalo *w*.

3.4.3. Valoración

Obviamente, el punto más fuerte de la arquitectura basada en servidores es la gran flexibilidad y libertad ofrecida al diseñador para configurar a su gusto el sistema y tratar de sacar el máximo provecho de él. En contra, la complejidad de los servidores es notablemente más alta por varios motivos:

- El hecho de disponer de múltiples opciones de planificación complica el análisis de planificación.
- Los servidores en sí son una estructura compleja, que necesitan soporte del núcleo de tiempo real para ser implementados, lo cual no está siempre disponible.
- Las tareas no corren en los servidores de manera transparente, es decir, hay que programarlas siendo consciente de que van a ejecutarse dentro de un servidor, lo cual, aumenta su complejidad.

Además de estos puntos, hay uno que cobra especial relevancia cuando nos introducimos en los sistemas de alta integridad, y es que la arquitectura basada en servidores por sí misma no ofrece mecanismos para corregir las violaciones de tiempo de ejecución que se puedan producir. Es decir, cuando un servidor agota su cuota, simplemente se queda suspendido hasta que dicha cuota es rellenada en algún momento posterior, lo cual, lejos de solucionar el problema, probablemente lo agrave ya que, si una tarea no cumple sus plazos seguramente se producirá una reacción en cadena que pondrá en peligro la integridad del sistema.

Por supuesto, estos mecanismos pueden ser incorporados al sistema como una capa más, pero sería deseable una arquitectura que agrupe todos estos módulos, de manera que la planificación, la monitorización del tiempo de ejecución y el tratamiento de fallos estén integrados en la propia arquitectura, simplificando el funcionamiento de la aplicación de usuario y por tanto facilitando la labor del diseñador del sistema. De esta manera, se reducirían sensiblemente los costes tanto directos (menor esfuerzo en el diseño), como indirectos, pues cuanto más sencilla sea la etapa de diseño, menor será el número de errores introducidos, simplificando la fase de integración.

3.5. Evaluación general y propuesta

Tras analizar los dos métodos más relevantes en la actualidad a la hora de construir sistemas particionados, en cada uno de ellos se ha destacado una serie de aspectos negativos, los cuales, consideramos que pueden y deben ser mejorados.

A grandes rasgos, la característica más destacable de ARINC 653 es el nivel de determinismo que proporciona, sin embargo, esto es posible gracias a una rigidez que penaliza sensiblemente la eficiencia del sistema. Por otra parte, la flexibilidad es precisamente la

mayor fortaleza de la arquitectura basada en servidores pero, este modelo no hace demasiado énfasis en la seguridad. Por estos motivos, uno de los objetivos de esta tesis es diseñar una nueva arquitectura en la que se combinen los puntos fuertes de los ejemplos evaluados, minimizando sus desventajas. Es decir, construir una arquitectura para soportar sistemas de tiempo real, flexible como para maximizar su eficiencia, a la vez que se proporcione un grado de seguridad suficientemente alto como para ser utilizado en sistemas de alta integridad.

El lenguaje de programación Ada en su versión 95 ya proporcionaba un buen número de herramientas para construir eficientemente sistemas de tiempo real, pero gracias a las mejoras que se han introducido en el nuevo estándar Ada 2005 (ver sección 2.3.2), se ha abierto una nueva posibilidad para llevar a cabo la planificación jerárquica en la que, tanto el planificador global como los planificadores locales, están integrados en un mismo entorno (Pulido et al., 2006). Esta tesis propone utilizar esta nueva técnica, denominada *arquitectura de bandas de prioridad*, que combina elementos de las anteriores, reforzando sus ventajas y minimizando sus debilidades. Así pues, las bandas de prioridad tienen grandes similitudes con la planificación basada en servidores, cuya principal ventaja es que se trata de un entorno mucho más flexible que ARINC 653, en el que realizar cambios es más sencillo y la comunicación entre particiones se puede realizar de una manera más fácil mediante el uso de objetos protegidos. Sin embargo, combina esta flexibilidad con los puntos fuertes de ARINC 653, que son la seguridad y sencillez, ya que:

- Los dos niveles de planificación se encuentran integrados, por lo que el núcleo de tiempo real que soporta la ejecución puede ser mucho más sencillo.
- Existen herramientas para monitorizar el tiempo de ejecución y programar temporizadores *hardware* que provoquen el desalojo inmediato de una tarea que sobrepasa su cuota.

Además, todos los elementos de la arquitectura propuesta están basados en el lenguaje Ada, todas las funciones que se utilizan están integradas en el propio lenguaje, lo cual facilita sensiblemente el proceso de desarrollo y aumenta la portabilidad del sistema, factor clave a la hora de reducir costes mediante la reutilización de componentes en proyectos diferentes.

Capítulo 4

ARQUITECTURA DE BANDAS DE PRIORIDAD

4.1. Introducción

Uno de los aspectos que más se ha cuidado en la elaboración de esta tesis ha sido la integración en proyectos más amplios de los avances que se han ido produciendo como consecuencia de nuestra investigación. De esta forma se evita obtener resultados que, aun resolviendo problemas técnicos concretos, son difíciles de llevar a la práctica por falta de una interfaz compatible con otros módulos o herramientas con los que interactuar.

Con este objetivo, todos los desarrollos que aquí se exponen han sido realizados conforme con los requisitos y restricciones definidas en el proyecto ASSERT (revisar apartado 1.3.2), asegurando así un buen nivel de visibilidad del trabajo realizado así como un excelente nivel de interacción con otros componentes.

La siguiente sección de este capítulo se dedica a explicar la arquitectura completa que se utiliza en ASSERT, capaz de dar servicio a sistemas de tiempo real empotrados y distribuidos con un alto grado de fiabilidad. A pesar de que muchos de los detalles que se facilitan están fuera del ámbito de esta tesis, se considera conveniente esbozar el marco general de la arquitectura, mostrando sus características principales con objeto de conocer mejor cómo se lleva a cabo la integración de todos los componentes.

Posteriormente, entrando ya en el radio de acción directo de esta tesis, la atención se centra sobre el detalle del núcleo de tiempo real y en la estructura de su planificador.

Más adelante se justificará la necesidad de mecanismos adicionales para proporcionar seguridad en el sistema. En primer lugar se trata el aislamiento temporal, tema central de la tesis, que consiste en garantizar que una aplicación no consumirá más tiempo de procesador que el previsto en tiempo de diseño, garantía que se implementa en tres pasos:

- Análisis estático, para comprobar en tiempo de diseño si un conjunto de tareas es planificable o no.
- Monitorización en tiempo de ejecución, para verificar que las premisas empleadas en el análisis estático se conservan en tiempo de ejecución y detectar cualquier anomalía.
- Tratamiento de fallos temporales, para dar una respuesta adecuada ante cualquier mal funcionamiento temporal.

Por último, aunque la tesis se centra en el aislamiento temporal, para cerrar el capítulo se indican las líneas a seguir para proporcionar aislamiento espacial, encargado de garantizar que una aplicación no va a invadir el espacio de memoria de cualquier otra.

4.2. Arquitectura completa de la máquina virtual

Dentro de la apuesta firme de la Comunidad Europea por mejorar la competitividad de la industria aeroespacial europea surgió el proyecto ASSERT, que centra su atención sobre los sistemas críticos embarcados, de tiempo real, distribuidos, fiables y basados en componentes.

Uno de los principales desafíos del proyecto ASSERT consiste en desarrollar una arquitectura con la que se puedan construir redes de sistemas empotrados, que sea capaz de ocultar la complejidad de las comunicaciones y de los cómputos inherentes a este tipo de sistemas, a la vez que provea una distribución efectiva y eficiente de los recursos, todo ello con un coste reducido.

Con este objetivo se ha definido una lista de requisitos que la máquina virtual debe cumplir. Estos son:

1. Soporte para diferentes niveles de criticidad: orientado a procesos de certificación, la máquina virtual debe proteger a las aplicaciones que se ejecuten en ella de errores

externos, evitando que aplicaciones certificadas a alto nivel se vean comprometidas por fallos de otras certificadas a más bajo nivel.

2. Soporte para la contención de fallos: en consonancia con el requisito anterior, la máquina virtual debe proporcionar aislamiento temporal y espacial para evitar la propagación de errores desde unas aplicaciones hacia otras.

3. Soporte para datos compartidos: todo ello con un control estricto, incluyendo tipado fuerte y mecanismos de control de acceso para evitar problemas derivados de la concurrencia.

4. Soporte para modelos de proceso complejos: debe tenerse en cuenta que se van a ejecutar aplicaciones heterogéneas, lo cual incluye: aplicaciones con varias tareas internas, aplicaciones distribuidas, activación de tareas periódicas y esporádicas, así como restricciones de precedencia y/o exclusión mutua.

5. Transparencia en la distribución: las aplicaciones deben tener la posibilidad de comunicarse entre sí independientemente del nodo en el que se encuentren, es decir, la infraestructura de comunicaciones debe ser transparente a las aplicaciones.

6. Puntualidad y predictibilidad: el comportamiento temporal debe ser analizable estáticamente y garantizado en tiempo de ejecución.

7. Confiabilidad: el sistema debe establecer sistemas de filtrado que garanticen que el flujo de información sea correcto.

La figura 4.1 muestra la arquitectura completa de la máquina virtual. Básicamente, la arquitectura está formada por tres componentes, configurados para operar sobre una plataforma *hardware LEON2* :

- Núcleo de tiempo real: soporta la ejecución de tareas de manera concurrente conforme al perfil de Ravenscar.
- Servicio de transferencia de mensajes: permite comunicaciones de bajo nivel entre los diferentes nodos.
- *Middleware*: facilita los servicios de distribución de manera transparente para las aplicaciones, que pueden estar en el mismo o en diferentes nodos.

Los siguientes apartados muestran una descripción de cada componente, siendo el núcleo de tiempo real el componente que merece más atención por ser el que incorpora la mayoría de los desarrollos elaborados en el marco de esta tesis.

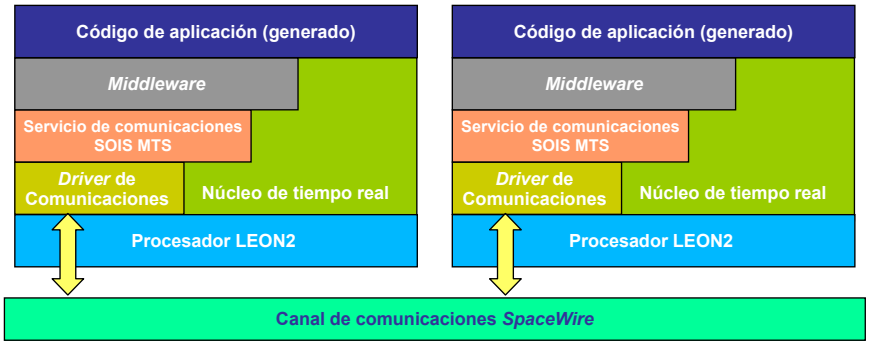

Figura 4.1: Arquitectura de la máquina virtual

4.2.1. *Middleware*

El componente *middleware* que forma parte de la máquina virtual está basado en PolyORB, producto desarrollado inicialmente por la *École Nationale Supérieure des Télécommunications* de París (Vergnaud et al., 2004). Desde un punto de vista funcional, es el encargado de asegurar que desde un nodo se puede servir una petición procedente de un nodo distinto, siendo por tanto el principal encargado de cumplir con el requisito número 5 *"transparencia en la distribución"*.

Para cumplir con su misión, se distinguen siete fases necesarias para llevar a cabo la comunicación entre diferentes aplicaciones. Esta secuencia es:

1. Direccionamiento: a cada entidad se le asigna un identificador global único.
2. Enlace: el *middleware* establece y mantiene asociaciones entre recursos que interactúan, permitiendo la comunicación.
3. Serialización: los parámetros de una petición son transformados de una manera adecuada para ser transmitidos a través de una red de comunicaciones.
4. Protocolo: el *middleware* implementa un protocolo para la transmisión de mensajes entre los diferentes nodos.
5. Transporte: se establece un canal de comunicaciones entre un nodo y un objeto para transmitir peticiones.
6. Activación: el *middleware* asegura que una entidad concreta está disponible para procesar una petición.

7. Ejecución: finalmente, se asignan recursos para procesar cada nueva petición.

4.2.2. Servicio de transferencia de mensajes

El servicio de transferencia de mensajes está basado en el protocolo SOIS-MTS y se encarga de dar el soporte de bajo nivel para las comunicaciones teniendo presente los requisitos definidos anteriormente. Junto con el componente *middleware*, se encarga de soportar los servicios de distribución transparente, teniendo muy en cuenta los requisitos número 2 *(contención de fallos)* y número 6 *(puntualidad y predictibilidad)*. Para ello, este componente implementa un servicio de filtrado por el que se regula la comunicación entre aplicaciones de distinto nivel de criticidad, asegurando así que un mensaje procedente de un nivel de certificación más bajo no comprometerá la integridad de una aplicación certificada a más nivel. Para cumplir con el requisito número 6, el servicio de transferencia de mensajes utiliza un protocolo determinista que permite dar una cota superior del tiempo necesario para llevar a cabo una comunicación. Además, implementa mecanismos para evitar sobrecargas que bloqueen el acceso al medio compartido.

4.2.3. Núcleo de tiempo real

La función del núcleo de tiempo real es dar soporte a la ejecución de aplicaciones con múltiples tareas en un sólo nodo. Como punto de partida se ha tomado el sistema de compilación *GNAT Pro for ERC32*, que contiene una evolución del núcleo de tiempo real *ORK*. La decisión se debe principalmente a que se puede sacar provecho de la experiencia acumulada a lo largo de los años por los miembros de nuestro grupo de investigación, ya que previamente se han desarrollado numerosos trabajos en torno a este núcleo de tiempo real.

Tal y como se introducía en la sección 2.3.4, ORK es un núcleo mínimo de altas prestaciones que proporciona soporte para ejecutar programas acordes con el perfil de Ravenscar de Ada y también para el lenguaje C. ORK está integrado con el compilador GNAT, siendo *GNAT Pro for ERC32*, desarrollada por la empresa AdaCore, la versión más moderna disponible a la fecha de comienzo de este trabajo. A lo largo del mismo, se ha desarrollado *GNATforLEON*, que es una evolución diseñada para operar sobre la plataforma *hardware* LEON2 (Urueña et al., 2007).

Dentro de la arquitectura propuesta en ASSERT, sobre el núcleo de tiempo real recae especialmente la responsabilidad de dar respuesta al requisito número 6 “puntualidad y

predictibilidad", así como dar soporte a los requisitos del 1 al 4 (soporte para diferentes niveles de criticidad, contención de fallos, diferentes modelos de proceso y de datos compartidos). El núcleo ORK está basado en el modelo de concurrencia restringido del perfil de Ravenscar, por ser un modelo de solvencia contrastada en sistemas de alta integridad que permite realizar un análisis formal de las propiedades temporales de un sistema utilizando técnicas de análisis de tiempo de respuesta. El modelo Ravenscar incluye un conjunto de tareas estáticas (son creadas en tiempo de elaboración y ejecutan un bucle infinito), y una sincronización restringida mediante objetos protegidos. Este modelo es suficientemente sencillo para ser soportado por un núcleo pequeño y que, por tanto, puede someterse a un proceso de certificación. A su vez, como consecuencia de la sencillez el núcleo es capaz de ofrecer un alto nivel de prestaciones, aumentando la velocidad de sistemas operativos tradicionales y disminuyendo considerablemente su huella en memoria, lo cual es especialmente relevante en sistemas empotrados.

Las propiedades del modelo Ravenscar dan soporte arquitectónico para cumplir con los requisitos 3, 4 y 6, sin embargo, no está contemplado el soporte en tiempo de ejecución para vigilar que todo transcurre dentro de los parámetros considerados durante el diseño del sistema. Asimismo, no hay ninguna referencia al aislamiento espacial, por lo que el modelo de Ravenscar debe ser ampliado para poder cumplir con todos los requisitos enumerados en la descripción de la arquitectura ASSERT. Consecuentemente, esta ampliación debe incorporar:

- Detección de sobrecargas mediante monitorización del tiempo de ejecución para garantizar aislamiento temporal entre aplicaciones.
- Garantía de aislamiento espacial entre aplicaciones.

La figura 4.2 muestra la arquitectura interna original del núcleo de tiempo real. Sus funciones pueden agruparse en varios grupos fundamentales:

- Gestión de tareas: incluyendo su creación, sincronización y planificación.
- Servicios temporales: principalmente el reloj de tiempo real y los retardos absolutos.
- Gestión de memoria: restringiendo la reserva dinámica de memoria al arranque del sistema, fase durante la cual se reserva espacio de pila para las tareas.
- Manejo de interrupciones: atendiendo a las restricciones especificadas por el perfil de Ravenscar.

junto con estos paquetes especialmente relevantes, el núcleo también incluye funciones para gestionar una línea serie además de otras funciones privadas que dan soporte a los paquetes enumerados anteriormente pero no son visibles desde el exterior.

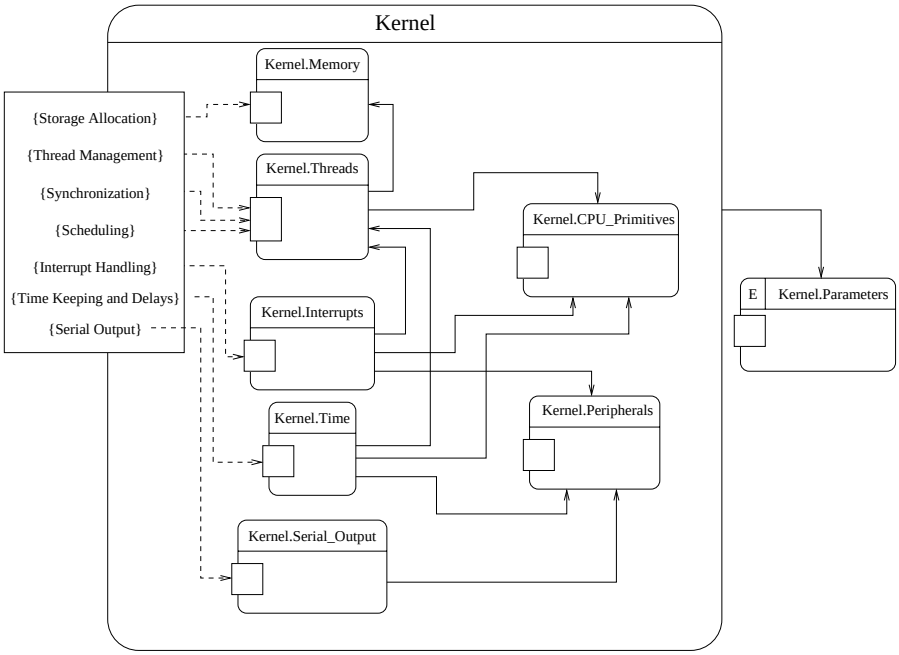

Figura 4.2: Estructura del núcleo ORK

4.3. Estructura del planificador

Dada la importancia que tiene la planificación en esta tesis, dentro de la arquitectura del núcleo real se le debe prestar una atención especial a la estructura del planificador, ya que ésta es una de las partes fundamentales de la propuesta que persigue mejorar el rendimiento que se puede obtener de las arquitecturas analizadas en el capítulo 3.

La *arquitectura de bandas de prioridad*, mostrada en la figura 4.3, está pensada para maximizar la flexibilidad manteniendo un nivel de seguridad tan alto como puedan proporcionar otras arquitecturas (e.g. ARINC 653). Puede observarse que la arquitectura de bandas de prioridad cuenta con un planificador global configurado para seguir una política de prioridades fijas. Además, se muestra cómo el rango global de prioridades se divide en una serie de bandas de prioridad disjuntas. Cada una de estas bandas cuenta con un

planificador local, que es el encargado real de despachar las tareas que están dentro de su rango. Para aumentar la flexibilidad del sistema, cada planificador local puede utilizar una política de planificación diferente (e.g. prioridades fijas, EDF, *Round-Robin*) y así adaptarse mejor a los requisitos de la aplicación.

Puede verse que, por su propia naturaleza, el planificador global siempre selecciona la banda de prioridad más alta que contiene al menos una tarea lista para ser ejecutada. Una vez seleccionada la banda de prioridad, el planificador local correspondiente, siguiendo su propia política de planificación, elige entre la cola de tareas listas para ejecutarse cuál es la que debe ocupar el procesador, resaltando la ventaja que supone que cada una de estas bandas puede estar configurada con una política de planificación diferente.

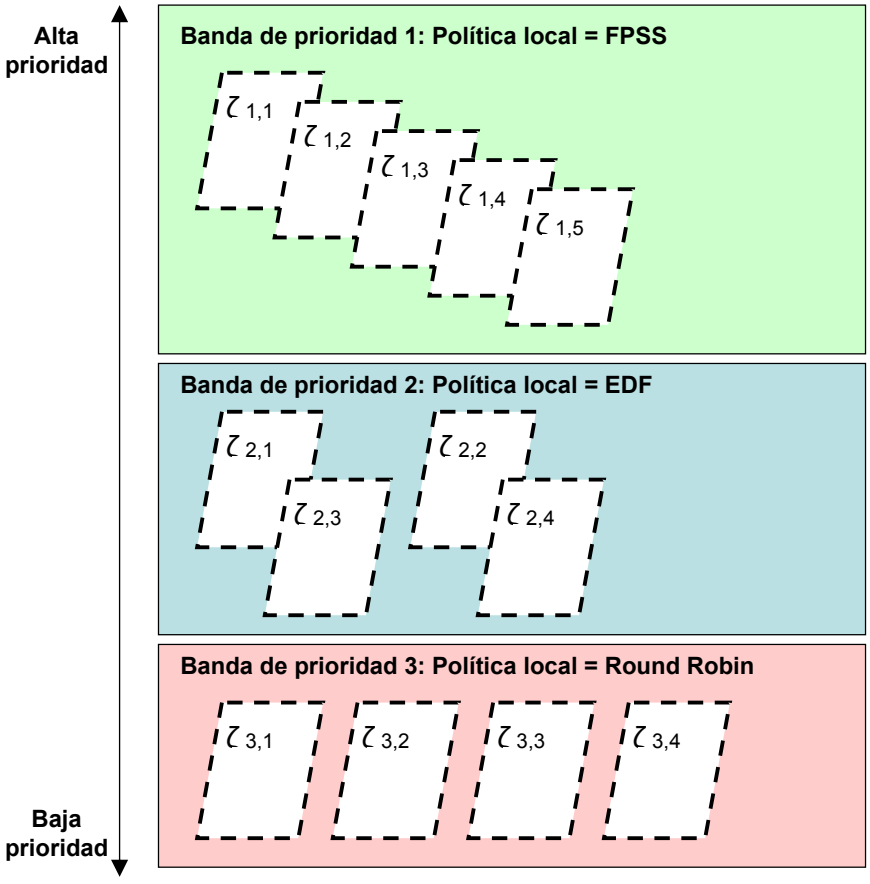

Figura 4.3: Arquitectura basada en bandas de prioridad

Con este modelo se puede pensar en una arquitectura de planificación jerárquica integrada, en la que cada banda de prioridad delimita una *partición virtual* en la que se puede

ejecutar una tarea (o conjunto de ellas) de manera segura, es decir, sin injerencias de las demás. Así, sería lógico pensar en un sistema compuesto por varias aplicaciones, en el que se defina una banda de prioridad para cada una de ellas, obteniendo así el deseado aislamiento entre aplicaciones.

Para poder llevar a cabo con éxito esta propuesta hay que tener en cuenta una serie de consideraciones que se van a abordar seguidamente. En primer lugar se debe tener presente que, si bien la flexibilidad de la que se hablaba viene dada por la propia naturaleza de la arquitectura, especialmente gracias a su planificador global basado en prioridades fijas que elimina las incómodas restricciones de la planificación estática, es necesario incorporar los mecanismos adecuados para proporcionar el nivel de seguridad deseado. Por ello, los siguientes apartados tratan los siguientes puntos:

- Análisis estático: en tiempo de diseño se pueden utilizar una serie de datos y premisas para pronosticar el comportamiento del sistema y así, comprobar a priori si un conjunto de tareas es planificable o no.
- Monitorización del tiempo de ejecución: para tener la certeza de que el resultado del análisis es fiable, en tiempo de ejecución se debe verificar que el comportamiento del sistema se ajusta a las premisas utilizadas en la fase de análisis y, en su caso, detectar cualquier anomalía, ya que de lo contrario el comportamiento del sistema podría volverse impredecible.
- Tratamiento de fallos temporales: en caso de que se produzca alguna anomalía es fundamental dar una respuesta adecuada para salvaguardar la integridad del sistema.
- Aislamiento espacial: análogamente al tiempo de ejecución, se deben tomar medidas para garantizar que los espacios de memoria asignados a cada partición son respetados.

4.4. Análisis estático

El primer paso para dotar al sistema de la seguridad necesaria es proporcionar *aislamiento temporal*, de manera que una tarea no pueda utilizar el procesador durante más tiempo del previsto y así evitar que tareas de menor prioridad se vean privadas del tiempo necesario para cumplir con sus plazos. Para ello hay que empezar por realizar un análisis estático o de tiempo de respuesta para comprobar si un conjunto de tareas con parámetros

conocidos es o no planificable, es decir, si el tiempo de respuesta de cada tarea es menor que su plazo límite.

La arquitectura basada en bandas de prioridad tiene un planificador global configurado para seguir una política de planificación con prioridades fijas, lo cual facilita el análisis en primera instancia. La mayor dificultad viene dada por el hecho de que las diversas bandas pueden seguir políticas distintas, de manera que el análisis es diferente según el sistema utilice unos protocolos u otros. En este trabajo se van a considerar solamente dos políticas de planificación a nivel local: *FPPS* y *EDF*.

El caso en el que todos los planificadores locales utilizan FPPS es el más sencillo, ya que al ser éste el protocolo utilizado también a nivel global, a pesar de ser un caso particular de planificación jerárquica, a efectos de realizar el análisis de planificación el sistema se comporta como uno basado en prioridades fijas sin planificación jerárquica, exactamente igual al caso visto en la sección 2.2.2, donde se mostraba que el tiempo de respuesta de una tarea en el peor caso posible viene dado por las expresiones 4.1 y 4.2:

$$w_i^{n+1}(q) = (q+1)C_i + B_i + \sum_{j \in hp(i)} \left\lceil \frac{w_i^n(q) + J_j}{T_j} \right\rceil C_j \tag{4.1}$$

$$R_i(q) = w_i^n(q) - q \cdot T_i + J_i \tag{4.2}$$

donde finalmente:

$$R_i = \max_{q=1,2,\ldots} R_i(q)$$

siendo el máximo valor de q que necesita ser probado el menor entero que verifica:

$$R_i(q) \leq T_i$$

El análisis de tiempo de respuesta resulta más complicado cuando se utilizan diferentes políticas de planificación a nivel local, es decir, cuando en al menos una de las bandas de prioridad las tareas son planificadas utilizando *EDF*. No obstante, el trabajo (González-Harbour and Palencia, 2003) resuelve el caso en el que se utiliza *EDF* localmente en un sistema planificado globalmente con prioridades fijas, lo que es de aplicación directa en la arquitectura de bandas de prioridad, por lo que se muestra a continuación la manera de llevar a cabo el análisis.

Una de las principales contribuciones de González y Palencia es que demuestran que: *"Una tarea τ_j planificada en una banda EDF con prioridad fija global $P_j > P_a$ produce la misma interferencia en τ_a que si dicha banda tuviera un planificador local configurado con FPPS"*. Es decir, a la hora de analizar la contribución al caso peor de una tarea τ_a causado por las bandas superiores, no importa si la política de planificación de dichas bandas es *EDF* o *FPPS* ya que en ambos casos tienen el mismo impacto. Por tanto, las ecuaciones 4.1 y 4.2 son válidas para analizar las tareas pertenecientes a bandas de prioridad configuradas con *FPPS* independientemente de la política utilizada en las bandas superiores.

En el caso del las tareas pertenecientes a bandas *EDF*, el análisis resulta un poco más complejo aunque puede ser igualmente llevado a cabo. A continuación se resume los pasos a seguir para realizar dicho análisis, calculando el tiempo de respuesta de una tarea τ_a perteneciente a una banda *EDF*.

El primer paso consiste en analizar la contribución de aquellas tareas con prioridades mayores que τ_a. Su contribución es máxima cuando son activadas al principio del periodo ocupado después de haber experimentado su máxima fluctuación (ver ecuación 4.3).

$$w_i(t) = \left\lceil \frac{t + J_i}{T_i} \right\rceil \cdot C_i \tag{4.3}$$

Posteriormente, se calcula la máxima longitud del periodo ocupado añadiendo la contribución de aquellas tareas con prioridad mayor o igual que τ_a, donde B_a^{ipc} considera el tiempo de bloqueo causado por tareas de menor prioridad:

$$L = B_a^{ipc} + \sum_{\forall i | P_i \geq P_a} \left\lceil \frac{L + J_i}{T_i} \right\rceil \cdot C_i \tag{4.4}$$

La complejidad reside en que el instante crítico puede ser distinto del punto inicial del periodo ocupado, por lo que es necesario analizar también aquellos instantes Ψ_x en los que el plazo de respuesta de τ_a coincide con el plazo de alguno de los trabajos con la misma prioridad τ_i donde $P_a = P_i$ (incluyendo la propia tarea bajo análisis). Se ha utilizado el término *trabajo* porque es posible que se produzcan varias activaciones de una misma tarea dentro del periodo ocupado y todas ellas deben ser tenidas en cuenta. Así pues, el tiempo de respuesta de la tarea τ_a viene dado por:

$$R_a = \text{máx}(R_a^A(p)) \tag{4.5}$$

$$\forall p = 1 \dots \left\lceil \frac{L + J_a}{T_a} \right\rceil, \forall A(\Psi_x) | \Psi_x \in \Psi^*$$

donde $A(\Psi_x)$ representa el punto de activación de de un trabajo cuyo plazo de respuesta expire en el instante Ψ_x y p es un número entero que representa el número de activación de una tarea dentro del periodo ocupado.

Para obtener los posibles valores de Ψ_x es necesario explorar todas las tareas con la misma prioridad que τ_a (incluyendo de nuevo la propia tarea τ_a:

$$\Psi = \bigcup \{(p-1) \cdot T_i - J_i + d_i\} \tag{4.6}$$

$$\forall p = 1 \dots \left\lceil \frac{L + J_i}{T_i} \right\rceil, \forall i | P_i = P_a$$

Y extraer aquellos valores pertenecientes al periodo:

$$\Psi^* = \{\Psi_x \in \Psi | (p-1) \cdot T_a - J_a + d_a \leq \Psi_x < p \cdot T_a - J_a + d_a\} \tag{4.7}$$

Entonces:

$$A(\Psi_x) = \Psi_x - [(p-1) \cdot T_a - J_a + d_a] \tag{4.8}$$

$$D^A(p) = A(\Psi_x) - J_a + (p-1) \cdot T_a + d_a \tag{4.9}$$

$$w_a^A(p) = B_a + p \cdot C_a + \sum_{\forall i \neq a | P_i = P_a} \text{mín} \left(\left\lceil \frac{w_a^A(p) + J_i}{T_i} \right\rceil, \left\lfloor \frac{J_i + D^A(p) - d_i}{T_i} \right\rfloor + 1 \right) \cdot C_i + \\ + \sum_{\forall j | P_j > P_a} \left\lceil \frac{w_a^A(p) + J_j}{T_j} \right\rceil \cdot C_j \tag{4.10}$$

Y finalmente:

$$R_a^A(p) = w_a^A(p) - A(\Psi_x) + J_a - (p-1) \cdot T_a \tag{4.11}$$

Este método permite realizar el análisis de tiempo de respuesta de un sistema con bandas de prioridad, a la par que resulta muy útil para decidir la configuración de los planificadores locales, ya que se puede comprobar fácilmente si se obtienen ventajas utilizando una política de planificación u otra.

4.5. Monitorización en tiempo de ejecución

A pesar de que en la sección anterior se ha explicado la manera de realizar un análisis estático para conocer en tiempo de diseño si un conjunto de tareas dado es viable o no, es decir, si todas las tareas van a ser capaces de terminar su trabajo antes de que expire su plazo de respuesta, no se puede confiar la seguridad del sistema solamente a las técnicas de planificación. Esta afirmación se basa en dos motivos:

- En primer lugar, la planificación se hace *a priori* tomando como válidos una serie de parámetros que pueden no cumplirse después en tiempo de ejecución, siendo dos los casos más habituales:
  - — Una tarea esporádica puede activarse más frecuentemente de lo esperado, por ejemplo, una pieza *hardware* defectuosa puede producir interrupciones continuamente.
  - — El peor tiempo de ejecución de una tarea puede ser peor de lo previsto.
- En segundo lugar, ya se ha comentado que no todas las aplicaciones son certificadas al mismo nivel, con lo que es posible que en un nodo conviva una aplicación certificada a un alto nivel, y por tanto muy fiable, con una certificada a más bajo nivel, con lo que no se puede confiar tanto en su corrección ya que su control de calidad ha sido menos exhaustivo y por tanto es más probable que contenga errores que pongan en peligro la integridad del sistema completo.

Por estos motivos, a pesar de que el análisis estático es un paso imprescindible para comprobar la viabilidad de un conjunto de tareas, en sistemas de alta integridad es igualmente importante asegurar que en tiempo de ejecución no se va a violar ninguna de las restricciones consideradas en el análisis.

El estándar Ada 2005 incluye en el propio lenguaje nuevos mecanismos de monitorización de tiempo de ejecución que permiten vigilar en todo momento que los cálculos realizados en la fase de diseño son respetados, así como tomar las medidas oportunas en

caso de que se produzca la vulneración de alguno de los supuestos en los que se basa el análisis. Los tres nuevos mecanismos que pueden ser utilizados en la arquitectura de bandas de prioridad son:

- Relojes de tiempo de ejecución: cada tarea lleva asociado un reloj que mide cuánto tiempo de procesador ha consumido una tarea desde su primera activación hasta el instante actual.
- Temporizadores de tiempo de ejecución: permiten asociarle a una tarea un temporizador cargado con un cierto valor de tiempo de ejecución, de manera que si dicho temporizador expira antes de ser desactivado o rearmado con un nuevo valor, se dispara una alarma en el sistema que permite ejecutar un procedimiento definido por el usuario para tomar las medidas oportunas.
- Cuotas de tiempo de ejecución para grupos de tareas o cuotas colectivas: de manera similar a los temporizadores de tiempo de ejecución, permiten armar un temporizador con un cierto valor, sin embargo, en lugar de estar asociadas a una sola tarea, este mecanismo se utiliza para vigilar el consumo de un grupo de ellas de manera conjunta.

Estas herramientas ofrecen diferentes soluciones para realizar la monitorización en tiempo de ejecución que se analizan seguidamente.

4.5.1. Detección simple

El mecanismo más sencillo es la detección simple. Consiste en utilizar los relojes de tiempo de ejecución tal y como muestra el listado 4.1, donde se aprecia la estructura de una tarea que realiza un cierto trabajo útil monitorizada por un reloj de tiempo de ejecución.

El listado muestra el cuerpo de la tarea, cuyo elemento principal es un bucle infinito en el que la carga útil está representada por la instrucción *Realizar_Trabajo_Util*. Cuando la tarea finaliza su trabajo útil se hace una llamada a la función *Leer_Reloj* (lectura del reloj de tiempo de ejecución), que devuelve la cantidad de tiempo de ejecución consumida por la tarea desde su creación. Al restar esta cantidad de la leída en la iteración anterior se obtiene la cantidad de tiempo de ejecución consumido por la tarea durante el último ciclo. Posteriormente, sólo queda comprobar si este resultado supera el peor tiempo de

Listado 4.1: Detección simple

```
task Deteccion_Simple is
begin
   loop
      -- Realizar_Trabajo_Util;
      Leer_Reloj;
      Comparar_dos_ultimas_lecturas;
      if Resultado > Limite then
         Disparar_Alarma;
      end if;
      delay until Proxima_Activacion;
   end loop;
end Deteccion_Simple;
```

ejecución previsto para la tarea, en cuyo caso se debe disparar una alarma indicando que la tarea se ha excedido en el uso de los recursos asignados y se deben tomar las medidas adecuadas.

La sencillez es el punto a favor de este método ya que introduce una sobrecarga muy ligera en el sistema a todos los niveles:

- A nivel de núcleo, porque las modificaciones que son necesarias para dar soporte a esta herramienta son mínimas.

- En tiempo de ejecución, pues el número de instrucciones extra que deben ejecutarse en cada iteración es reducido porque simplemente hay que realizar una lectura del reloj de tiempo real para comprobar el tiempo de cómputo consumido

Por el contrario, el gran inconveniente de este método es que la detección se realiza *a posteriori*, es decir, la comparación entre el tiempo consumido y el máximo admitido se hace una vez que la tarea ha finalizado su trabajo. Por consiguiente, si la tarea se ha excedido en el uso del procesador, el daño ya está causado y es posible que alguna tarea de menor prioridad no haya completado su trabajo a tiempo, lo cual es una amenaza para la integridad del sistema.

Se puede concluir por tanto que el problema de la *detección tardía* inherente a este mecanismo es demasiado grave como para confiarle en exclusiva la seguridad de un sistema de tiempo real crítico.

4.5.2. Detección inmediata

Un mecanismo basado en una idea similar pero más sofisticado que el anterior es el que denominamos *detección inmediata*. El listado 4.2 presenta su algoritmo.

Listado 4.2: Detección inmediata

```
task Deteccion_Inmediata is
begin
   loop
      Armar_Temporizador;
      -- Realizar_Trabajo_Util;
      Desarmar_Temporizador;
      delay until Proxima_Activacion;
   end loop;
end Deteccion_Inmediata;
```

Puede observarse de nuevo que el elemento fundamental del cuerpo de la tarea es un bucle infinito que realiza un cierto trabajo útil. En este caso, justo antes y justo después de realizar dicho trabajo se hace una llamada a las funciones *Armar_Temporizador* y *Desarmar_Temporizador* respectivamente. El efecto es el siguiente; cuando la tarea comienza a ejecutarse por primera vez después de cada activación, se carga un temporizador con una cuota igual al tiempo de ejecución en el peor caso posible de la tarea; una vez montado este temporizador, la tarea realiza su trabajo útil a la vez que la cantidad de tiempo de ejecución cargada en el temporizador va disminuyendo (nótese que si la tarea es desalojada del procesador, el temporizador es detenido hasta que la tarea en cuestión vuelve a ocupar el procesador). Si la tarea termina su labor a tiempo, se ejecutan las instrucciones *Desarmar_Temporizador* (que desconecta el temporizador) y *Delay until*, por lo que la tarea queda suspendida hasta el instante *Proxima_Activacion*. Si por el contrario, la tarea supera el tiempo de ejecución previsto, el temporizador expira antes de ser desmontado, disparando una alarma que debe ser debidamente tratada (ver apartado 4.6).

La diferencia fundamental entre los dos métodos expuestos hasta el momento radica en que, mientras que en el primero, la detección del problema puede retrasarse *ad infinitum* porque el trabajo útil de la tarea no es interrumpido, con el mecanismo de detección inmediata el temporizador provoca una interrupción justo en el instante en el que se ha consumido la cantidad de tiempo de ejecución especificada por lo que, tomando las medidas de contención adecuadas, se puede garantizar la integridad de las tareas de menor prioridad, cuya ejecución era desplazada utilizando el mecanismo de detección simple.

Por contra, la detección inmediata requiere el uso de temporizadores de tiempo de ejecución, que producen una sobrecarga mayor en el sistema ya que, tanto para armar como para desarmar un temporizador hay que realizar una serie de operaciones a nivel de *hardware* que la aumentan. Además se deben introducir más operaciones en cada cambio de contexto. En el capítulo 6 se ofrecen datos cuantitativos concretos que facilitan la comparación entre ambos métodos.

4.5.3. Cuota colectiva

Una última variante para monitorizar el tiempo de ejecución es posible gracias a las cuotas colectivas, que funcionan de una manera similar a los temporizadores de tiempo de ejecución salvo por dos hechos:

- No contabilizan el consumo de una sola tarea sino el de un grupo de ellas.
- A diferencia de los temporizadores, en los que cada vez que se quiere monitorizar una sección de código se arma un temporizador con un valor concreto, las cuotas colectivas son inicializadas una sola vez con una cierta cantidad, que va siendo consumida por las diferentes tareas asociadas y que debe ser rellenada siguiendo algún algoritmo.

El hecho de que contabilicen el consumo de un grupo de tareas conjuntamente hace de esta variante una opción muy interesante en aquellos casos en los que no es especialmente relevante el consumo de una tareas por sí misma sino el consumo de una aplicación completa. Por ejemplo, esta situación se da cuando el diseñador del sistema quiere equilibrar el uso del procesador entre varias aplicaciones, asegurándose de que ninguna de ellas utiliza más de un cierto porcentaje del tiempo de procesador, sin tener que descender al nivel de tarea y comprobar los tiempos de ejecución de cada una de ellas. De hecho, la detección de errores temporales mediante cuotas colectivas no permite saber a priori cuál ha sido la tarea que ha provocado la sobrecarga, ya que puede ser la tarea que se esté ejecutando en el instante en que salta la alarma, o bien puede haber sido provocada por una tarea que se ejecutó con anterioridad y que consumió más tiempo del previsto, dejando demasiado poco restante para las demás tareas del grupo.

Las cuotas colectivas son una herramienta útil a la hora de implementar una arquitectura basada en servidores como los mostrados en el capítulo anterior. Sin embargo, utilizar esta funcionalidad del lenguaje Ada 2005 para monitorizar el consumo de CPU y garantizar aislamiento entre particiones no es trivial.

Parte del trabajo de investigación realizado en esta tesis ha consistido en evaluar diferentes algoritmos en los que se utilizan cuotas colectivas para controlar el tiempo de ejecución de varias tareas de manera conjunta. Dicho con otras palabras, se ha buscado un método que permitiera balancear el consumo de tiempo de procesador entre varias tareas sin invalidar el análisis estático presentado en la sección 4.4. Los resultados obtenidos no han sido satisfactorios ya que los algoritmos probados tendían en algunos casos hacia resultados optimistas (detecciones tardías), o bien pesimistas (falsos positivos).

El punto clave de un algoritmo que utilice cuotas colectivas está en el modo de recargar la cuota. Es decir, definir en qué momento debe hacerse la recarga y qué cantidad recargar. En el caso que nos ocupa, un sistema de tiempo real crítico basado en el perfil de Ravenscar, la opción más razonable es la más restrictiva desde el punto de vista temporal. Para ello, cada cuota colectiva debe ser inicializada y rellenada con la menor cantidad de tiempo de ejecución posible y además, el momento de la recarga debe posponerse hasta el último instante.

Listado 4.3: Detección mediante cuota colectiva

```
task Cuota_Colectiva is
begin
   loop
      -- Realizar_Trabajo_Util;
      Calcular_Tiempo_Consumido;
      delay until Proxima_Activacion;
      Rellenar_Cuota (Tiempo_Consumido);
   end loop;
end Cuota_Colectiva;
```

Una posibilidad estudiada se muestra en el listado 4.3. La cuota colectiva se crea e inicializa fuera de la tarea, con un valor inicial igual a la suma de los WCET de todas las tareas del grupo. El listado muestra como en cada activación, la tarea entra en su bucle principal, de manera semejante a lo visto en los ejemplos anteriores, en el que se realiza el trabajo útil de la tarea. Con objeto de cumplir la premisa establecida anteriormente, según la cual, la cuota debe recargarse con la menor cantidad de tiempo de ejecución y lo más tarde posible, una vez realizado el trabajo útil se calcula cuanto tiempo se ha consumido en esta iteración, recargando solamente esta cantidad al inicio de la siguiente iteración (primera instrucción después del *delay until*).

Este algoritmo no es válido ya que podría darse la situación mostrada en la figura 4.4. Una cuota monitoriza el consumo de las tareas $\tau_{1,1}$ y $\tau_{1,2}$. En la primera activación se ini-

cializa la cuota y se consume a medida que alguna de las dos tareas ocupa el procesador. En su primera activación, $\tau_{1,2}$ termina su trabajo en menos tiempo que su WCET estimado, por lo que queda cuota remanente. En la siguiente activación de $\tau_{1,1}$, utiliza dicho remanente. El problema se genera porque dicho remanente no se utiliza una sola vez (lo cual sí sería un balanceo correcto), sino que se utiliza también en la tercera y sucesivas aplicaciones, incumpliendo las premisas del análisis estático y provocando que la tarea $\tau_{2,1}$ incumpla su plazo de respuesta.

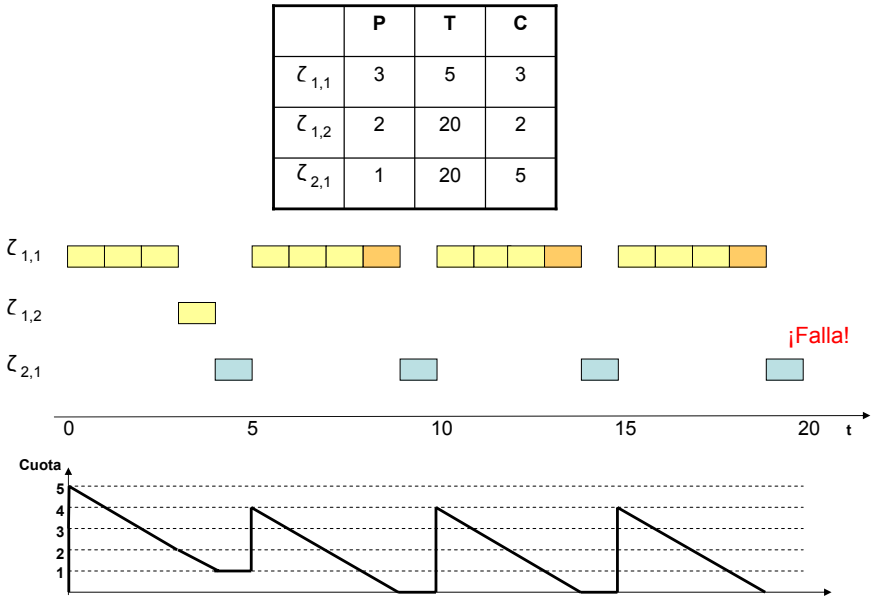

Figura 4.4: Fallo del algoritmo por exceso de optimismo

Para solucionar este problema, se puede pensar en modificar ligeramente el algoritmo, de manera que la cantidad a recargar no sea el tiempo consumido en la ejecución anterior cuando éste haya sobrepasado el WCET de la tarea. Es decir, la cantidad *R* a recargar sería:

$$R = \text{mín}(WCET, tiempo\_consumido)$$

Este algoritmo tampoco da resultado debido a que en esta ocasión el exceso de pesimismo puede dar lugar a falsas alarmas (ver figura 4.5). La causa es que cuando una tarea excede su WCET, aún balanceándolo con otra tarea de su grupo, como ocurre con las tareas $\tau_{1,1}$ y $\tau_{1,2}$ en la figura 4.5, ese exceso se traduce en una pérdida en el saldo de la cuota colectiva que no se recupera nunca. Por este motivo, cuando las tareas se acercan de nuevo a su WCET, se producen falsos positivos.

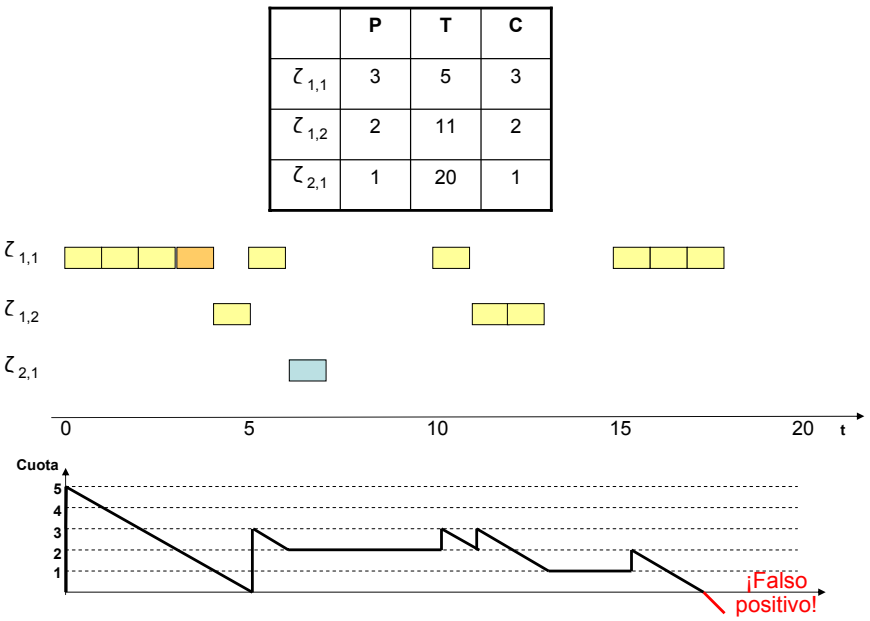

Figura 4.5: Fallo del algoritmo por exceso de pesimismo

En definitiva, debe continuarse el análisis de algoritmos que permitan equilibrar el consumo de tiempo de ejecución entre varias tareas. De momento, posponemos la utilización de cuotas colectivas y recomendamos el uso del mecanismo de la detección inmediata para monitorizar el comportamiento en tiempo de ejecución.

4.5.4. Cumplimiento del periodo mínimo entre activaciones esporádicas

Tal y como se ha apuntado previamente, otra de las fuentes de fallos temporales viene dada porque el análisis estático se hace teniendo en cuenta que las tareas esporádicas respetan un cierto intervalo entre dos activaciones consecutivas, con lo que el análisis de tiempo de respuesta establece como premisa que el periodo mínimo de estas tareas es igual a dicho intervalo. Sin embargo, en la práctica esta premisa puede incumplirse debido a múltiples circunstancias, por ejemplo, un fallo *hardware* en un sensor puede hacer que esté constantemente produciendo interrupciones (*babbling idiot failure*), que típicamente dispararán una tarea esporádica mucho más frecuentemente de lo esperado. Por este motivo, tan importante como vigilar que el tiempo de ejecución real no exceda del previsto, es también forzar que se cumpla el tiempo mínimo entre dos activaciones

consecutivas de una tarea esporádica.

Listado 4.4: Cumplimiento del periodo mínimo entre dos activaciones consecutivas

```
task Esporadica is
begin
   loop
      Esperar_Evento;
      -- Realizar_Trabajo_Util;
      delay until Proxima_Activacion;
   end loop;
end Esporadica;
```

El listado 4.4 muestra el sencillo algoritmo que debe seguirse para proteger el sistema de este error. Puede verse que en cada iteración la ejecución se queda bloqueada en dos puntos distintos: en primer lugar, esperando a que se produzca el evento que dispara la tarea esporádica. Cuando esto sucede, se realiza el trabajo útil y, seguidamente, se ejecuta una instrucción *delay until* que deja la tarea suspendida hasta que transcurra el intervalo mínimo. De esta manera, aunque se den los eventos demasiado próximos entre sí, al estar suspendida, la tarea no podrá atenderlos hasta su debido momento. Como se explicará más adelante, este algoritmo puede ser perfeccionado para tener en cuenta si se han producido intentos de disparar la tarea demasiado frecuentemente y así poder actuar en consecuencia.

4.5.5. Vigilancia del plazo de respuesta

A pesar de que el uso de los mecanismos expuestos hasta el momento, ya sea individualmente o combinando varios de ellos, garantizan con un alto grado de fiabilidad que el uso del procesador se va a ajustar a las premisas establecidas en tiempo de diseño, es conveniente vigilar que los plazos de respuesta de las tareas son finalmente cumplidos para poder detectar la violación de uno de estos plazos justo en el momento en que se produce.

Una de las novedades que introduce Ada 2005 son los eventos programados (*timing events*). Este mecanismo permite al programador definir un procedimiento para que sea ejecutado en un cierto instante de manera sencilla, introduciendo sólo una ligera sobrecarga, ya que no hay necesidad de utilizar una tarea.

La vigilancia del cumplimiento de plazos de respuesta puede hacerse mediante even-

Listado 4.5: Vigilancia del plazo de respuesta

```
task Monitored_Deadline is
begin
   loop
      Cargar_Evento_Programado;
      delay until Proxima_Activacion;
      -- Realizar_Trabajo_Util;
   end loop;
end Monitored_Deadline;
```

tos programados tal y como muestra el listado 4.5. Partiendo de que existe un procedimiento definido por el diseñador del sistema en el que se ejecutan las acciones necesarias en caso de que una tarea no cumpla su plazo de respuesta, el resto del algoritmo es muy sencillo. Tomando como base el pseudo-código habitual en el que el cuerpo de la tarea está basado en un bucle infinito que contiene un cierto trabajo útil, en este caso se trata de añadir una instrucción que carga un evento programado que debe dispararse justo en el momento en el que cumpla el plazo de respuesta de la próxima iteración del bucle. Así pues, en caso de que la tarea no realice su trabajo útil antes del tiempo previsto, se ejecutará el procedimiento definido por el diseñador del sistema. En caso contrario, cuando la tarea sí cumple su objetivo a tiempo, se ejecuta de nuevo la instrucción (*Cargar_Evento_Programado*), que sobreescribe el evento programado, modificando así el instante en el que debería saltar el manejador, pasando a ser la *deadline* de la próxima interacción del bucle en lugar de la actual.

4.6. Tratamiento de fallos temporales

Después de detectar un fallo temporal, el siguiente paso es dar una respuesta adecuada para minimizar su impacto y preservar la integridad del sistema (Pulido, Urueña, Zamorano and de la Puente, 2007). A continuación se muestran las acciones más lógicas que se pueden llevar a cabo. La elección de unas u otras depende de varios factores, tales como el estado global del sistema o la criticidad de la aplicación que provoca el fallo.

4.6.1. Notificación del fallo

La primera acción, y la menos intrusiva, es notificar y registrar el error producido, identificando la aplicación, y si es posible, la tarea que se ha excedido en el uso de los recursos. Obviamente, esta acción, que ni siquiera puede ser calificada como *correctora*, ya que la aplicación defectuosa continúa haciendo uso del procesador, solamente es válida en rangos de muy baja criticidad, donde un mal funcionamiento no pone en peligro la integridad del sistema en general o, durante las diferentes fases de pruebas en las que el objetivo sea simplemente recoger la mayor información posible acerca del comportamiento temporal de las aplicaciones. En cualquier otro caso es necesario ir más allá, puesto que resulta inadmisible que una tarea monopolice el uso del procesador.

4.6.2. Algoritmo de la segunda oportunidad

La segunda estrategia a seguir recibe el nombre de *algoritmo de la segunda oportunidad*. Éste es un mecanismo algo más intrusivo que el anterior ya que modifica ligeramente los parámetros que intervienen en la planificación. Consiste en que, cuando se detecta un exceso en el consumo de tiempo de ejecución, antes de tomar medidas de mayor calado como las que se muestran posteriormente, se alarga la cuota de la tarea problemática (o en su caso de la aplicación), haciendo uso del tiempo que el procesador está inactivo, dando de esta manera una segunda oportunidad a la tarea para que cumpla su objetivo, de ahí el nombre del algoritmo. Dicho de otra forma, el algoritmo de la segunda oportunidad utiliza el tiempo que el procesador permanece ocioso para intentar resolver situaciones de sobrecarga.

El uso de este algoritmo puede justificarse porque básicamente hay dos motivos por los que una tarea puede sobrepasar su tiempo de ejecución:

- Debido a algún problema grave, una tarea puede necesitar una cantidad de tiempo de ejecución muy superior al estimado, es posible incluso que sea infinito (por ejemplo si ha entrado en un bucle sin fin), lo cual puede ocasionar unos efectos colaterales cuyo alcance pone en peligro la integridad del sistema en general.
- Debido a un error leve en la estimación del peor caso de tiempo de ejecución se produce una sobrecarga puntual.

El primer caso de fallo es muy grave porque tiene implicaciones tanto para la tarea defectuosa como para las demás, ya que:

- La tarea defectuosa probablemente no será capaz de completar su trabajo, ni siquiera después de aumentar su cuota.

- Las tareas de menor prioridad no conseguirán hacerse con el procesador por lo que tampoco serían capaces de terminar su trabajo a tiempo.

- Las tareas de mayor prioridad también pueden verse afectadas por el mal funcionamiento, por ejemplo, si hacen operaciones tomando como entrada algún valor que debería ser actualizado por la tarea defectuosa.

En el segundo caso, sin embargo, cabe la posibilidad de que esta situación anómala se dé en casos muy aislados o incluso de que ocurra una vez y no vuelva suceder en el futuro, por lo que conceder a la tarea una prórroga para que intente terminar su trabajo puede resolver el problema sin necesidad de tomar decisiones más drásticas que afecten seriamente a la funcionalidad del sistema. En otras palabras, si el análisis muestra que hay un tiempo en el que el procesador permanece inactivo, se puede utilizar esta capacidad de cómputo remanente para concederle a la tarea una segunda oportunidad de terminar su trabajo y así comprobar si la sobrecarga ha sido transitoria o, si por el contrario, es suficientemente grave como para ir un paso más allá en las acciones correctivas a tomar para preservar la integridad del sistema.

Cálculo de la cuota extraordinaria

Para utilizar el algoritmo de la segunda oportunidad, el primer paso consiste en calcular la longitud de la cuota, que de manera extraordinaria, se le asigna a una tarea.

En ciertos tipos de sistemas, como pueden ser algunos de tiempo real flexible, o ciertas arquitecturas basadas en reservas, es habitual que se agoten las cuotas; en esos casos no se trata de un mal funcionamiento, simplemente es un evento más que provoca ciertas reacciones en el sistema, de la misma forma que lo haría un usuario pulsando un botón. Sin embargo, la arquitectura basada en bandas de prioridad está pensada para sistemas de tiempo real estricto, en los que el hecho de que una tarea agote su tiempo de cómputo no entra dentro de la normalidad; es una excepción que debe ser tratada. Como se ha expuesto en el apartado anterior, el objetivo de utilizar el algoritmo de la segunda oportunidad es proporcionar un tiempo extra para intentar que la tarea cumpla su misión ya que, de conseguirlo, el sistema podría continuar operando en modo normal, sin necesidad de poner en práctica medidas complejas.

Hay diferentes algoritmos que regulan el uso de este tiempo extra (véanse como ejemplo los trabajos (Davis et al., 1993; Burns and Wellings, 2001)). En el caso que nos atañe hay que tener en cuenta sus circunstancias particulares. Por un lado, agotar una cuota es un hecho excepcional. Por otro lado, si la tarea no termina su trabajo las consecuencias pueden ser muy graves. Por tanto, la opción más interesante es tratar de darle a la tarea defectuosa, de una sola vez, el mayor tiempo de cómputo que sea posible manteniendo el conjunto de tareas planificable, es decir, sin afectar a las tareas de menor prioridad.

Intentar calcular en tiempo de ejecución la longitud de la cuota extra con objeto de llevar al extremo la premisa de maximizar su valor resultaría contraproducente, ya que se trata de un cálculo muy pesado que no haría sino complicar aún más la situación de sobrecarga en la que se encuentra el sistema. Por consiguiente, la opción más conveniente es hacer el cálculo en tiempo de diseño a pesar de que se obtenga un valor más pesimista que el que se obtendría al hacer el cálculo dinámicamente.

El cálculo a realizar consiste en averiguar cuánto tiempo de procesador puede consumir una tarea sin provocar que las demás incumplan su plazo de respuesta. Para poder resolver correctamente esta cuestión hay que modificar ligeramente el análisis estático propuesto en la sección 4.4. El listado 4.6 muestra el algoritmo necesario. El primer paso es analizar si el conjunto de tareas es planificable utilizando los parámetros originales. En caso afirmativo, se itera el análisis aumentando el tiempo de cómputo (C_i) de una tarea τ_i hasta averiguar cuál es el máximo valor que puede tomar manteniendo el conjunto planificable (i.e. el máximo valor que permite asegurar que el tiempo de respuesta de cada tarea es menor que su *deadline*). Repitiendo esta operación para todas las tareas del conjunto se obtiene, para cada una de ellas, cuál es la longitud máxima de la cuota extraordinaria que se puede utilizar en caso de sobrecarga.

El cuadro 4.1 muestra un ejemplo sencillo que ilustra el funcionamiento del algoritmo. Sea un sistema con 3 tareas (τ_1, τ_2 y τ_3), con los periodos, prioridades y tiempos de cómputo en el caso peor mostrados. Aplicando el algoritmo anteriormente expuesto se obtiene los valores de la cuota extraordinaria que se podría utilizar en caso de que un temporizador de tiempo de ejecución lanzase una alarma. Lógicamente, el valor de la cuota extraordinaria crece de manera inversamente proporcional con las prioridades de las tareas, puesto que a menor prioridad menor riesgo existe de bloquear la ejecución de aquellas tareas que gozan de una prioridad aún menor.

Listado 4.6: Cálculo de la cuota extraordinaria

```
Planificable : Boolean := False;
Cuota_Extra  : Array (1 .. Max_Tasks) of CPU_Time;

Analisis_Tiempo_Respuesta (Planificable);

if Planificable then
   for i in 1 .. Max_Tasks loop
      while Planificable loop
         Aumentar C_i;
         Analisis_Tiempo_Respuesta (Planificable);
      end loop;
      Ajustar_Valor_Maximo_C_i;
      Cuota_Extra (i) := Valor_Maximo_C_i - C_i_inicial;;
   end loop;
else
   null; -- Conjunto inicial no planificable!!
end if;
```

Cuadro 4.1: Algoritmo de la segunda oportunidad

Tarea	Periodo	Prioridad	WCET	Cuota extraordinaria
τ_1	10	3	3	2.5
τ_2	20	2	5	5
τ_3	30	1	4	7

Utilización de la cuota extraordinaria

Tras explicar cómo se calcula el valor de la cuota extraordinaria que se utiliza en el algoritmo de la segunda oportunidad, inmediatamente surgen dos preguntas: ¿Se puede utilizar la cuota extraordinaria varias veces? Y, en caso afirmativo ¿hay que respetar algún intervalo mínimo entre dos utilizaciones consecutivas?

La respuesta es afirmativa en ambos casos. Por supuesto, se puede utilizar la cuota extraordinaria en múltiples ocasiones, siempre y cuando se cumplan las condiciones iniciales con las que su valor fue calculado. Nótese que el cálculo se realizaba alargando el tiempo de cómputo de la tarea afectada manteniendo el de las demás constantes, por lo que si se intenta alargar el tiempo de CPU concedido a una tarea inmediatamente después de que otra tarea haya utilizado su prórroga, muy probablemente alguna tarea perdería su plazo. Por tanto, para estar seguros de que no se violan las condiciones de análisis cada

vez que se utilice la cuota extraordinaria habrá que dejar transcurrir un hiperperiodo antes de volver a utilizarla.

4.6.3. Cambio de modo

Una última estrategia es posible y necesaria para tratar un fallo temporal. Cuando la tarea no es capaz de finalizar su trabajo a tiempo, ni siquiera después de alargar su cuota mediante el algoritmo de la segunda oportunidad, podría poner en serio peligro la integridad del sistema, por tanto, es necesario ir más allá y emprender acciones correctoras que limiten los efectos del fallo temporal y no ponga en riesgo la totalidad del sistema dejando tareas incontroladas. Para ello, lo más apropiado es efectuar un cambio de modo de funcionamiento en el sistema, siendo lo recomendable:

- Iniciar una secuencia de parada segura, en la que los diferentes servicios se van apagando para llevar al sistema a un estado seguro.
- Cambiar a un modo restringido en el que el sistema puede continuar operando pero sólo con una parte de su funcionalidad activa.
- Reiniciar el sistema, siempre y cuando el tiempo necesario para llevar a cabo esta operación y durante el cual el sistema quedaría sin gobierno, sea asumible.

Estas opciones son ejemplos clásicos de acciones de recuperación. Por supuesto, también existen métodos para garantizar que el sistema puede seguir operando con toda su funcionalidad disponible, sin embargo, para ello es necesario disponer de redundancia tanto *hardware* como *software*, lo cual no es objeto de esta tesis por lo que esta opción no será considerada de aquí en adelante.

Parada segura

El mecanismo de parada segura es el más sencillo de realizar ya que puede ser implementado en Ada 2005 mediante la ejecución de un *manejador de últimas voluntades*, incluido en el perfil de Ravenscar, que debe ejecutar todas las instrucciones necesarias para llevar el sistema hasta la situación segura. El problema es que no todos los sistemas pueden ser apagados. Por ejemplo, en caso de que se detecten anomalías graves en un tren, detenerlo es la opción más segura para los pasajeros. Obviamente, no se podría hacer lo

mismo con un avión. No obstante, lo que sí es posible prácticamente en todos los casos es hacer una parada segura de algún subsistema.

Modo restringido

Otra posibilidad consiste en poner en *cuarentena* a la tarea problemática (aquella que provocó que se disparara una alarma), de manera que no pueda continuar bloqueando al resto de tareas. Para ello se pueden ejecutar dos acciones:

- Reducir la prioridad de la tarea a la mínima del sistema, así sólo se ejecutaría en *segundo plano*, es decir, ocuparía el procesador única y exclusivamente cuando ninguna otra tarea estuviese activa.
- Abortar la tarea. Puesto que reduciendo su prioridad a la mínima es improbable que pueda cumplir su plazo, se puede optar directamente por abortar la tarea.

Ambas soluciones son una fuente de indeterminismo, por lo que su comportamiento temporal puede ser difícil de analizar, de hecho, tanto cambiar la prioridad de una tarea como abortarla, son instrucciones prohibidas por el perfil de Ravenscar. Además, cualquier dependencia funcional o sincronización con la tarea defectuosa debe ser tenida en cuenta en estas situaciones porque se pueden producir efectos colaterales no deseados. Por ejemplo, una tarea podría permanecer suspendida *ad infinítum* esperando en una barrera que debía ser levantada por la tarea defectuosa, o bien podría generar continuamente datos erróneos debido a parámetros que nunca son actualizados porque esta labor le correspondía a la tarea puesta en cuarentena. Por consiguiente, eliminar sin más una tarea de la planificación es una opción aceptable si, y solo si, desempeña una función prescindible y no hay ninguna otra tarea cuyo comportamiento dependa de ella.

Una acción más plausible es realizar un cambio de modo operacional, consistente en mantener operativos sólo ciertos servicios. De este modo se puede aligerar la carga del procesador y superar la situación de sobrecarga, manteniendo intactas las funciones más importantes del sistema y descartando aquellas más prescindibles. Uno de los principales problemas viene dado por el número de modos que se deben incluir en el sistema con vistas a afrontar todos los fallos potenciales que se puedan producir. Dado que un sistema típico puede contener en total varias docenas de tareas está claro que no es viable incluir el mismo número de modos de funcionamiento alternativos con objeto de cubrir un fallo en cada tarea de manera individualizada. Una solución más realista consiste en proveer un solo modo degradado, en el que se llevan a cabo las funciones vitales del sistema.

Un cambio de modo supone una profunda alteración en la configuración del sistema. El habitual escenario estático que propone el perfil de Ravenscar es alterado y por tanto puede ser complicado llevar a cabo el necesario análisis temporal, especialmente si se utilizan prioridades dinámicas, transferencia asíncrona de control o se abortan tareas, motivo por el cual, todas estas acciones están excluidas del perfil. No obstante, es posible implementar cambios de modo de una manera compatible con el perfil como ya mostraron (Alonso and de la Puente, 2001). La esencia del método propuesto en ese estudio consiste en separar las *tareas* de los *trabajos*, de manera que cada tarea tiene la posibilidad de ejecutar un trabajo distinto en cada iteración, dependiendo del modo operacional en que se encuentre el sistema. Esta forma de llevar a cabo un cambio de modo sí es analizable y, por tanto, válida para ser usada en sistemas de tiempo real críticos.

Tomando como punto de partida esta idea, proponemos un mecanismo alternativo para implementar un cambio de modo. Lo primero que se ha de tener en consideración es que cuando una tarea ocupa el procesador durante demasiado tiempo, todas aquellas tareas de menor prioridad quedan bloqueadas. Tal y como se ha comentado a lo largo de esta sección, no está permitido hacer uso de prioridades dinámicas ni abortar tareas, por tanto, si la tarea defectuosa, por ejemplo, ha entrado en un bucle infinito, no hay ninguna manera de rescatarla y, lo que es peor, no hay forma de evitar el bloqueo de todas las tareas que tengan una prioridad menor que la tarea defectuosa. Por consiguiente, los trabajos que tienen que ser ejecutados en el modo degradado deben estar dentro de tareas con un nivel de prioridad mayor que el de la tarea problemática.

Puesto que a priori no se sabe qué tarea va tener un comportamiento erróneo, es imposible conocer en tiempo de diseño a partir de qué nivel de prioridad van a quedarse bloqueadas las tareas. Por tanto, los trabajos a ejecutar en el modo degradado no pueden compartir las mismas tareas que aquellos que deben ser ejecutados en el modo operacional normal, sino que deben estar situados en una banda de prioridad superior para evitar posibles bloqueos. Dicho de otra forma, sean n_1 y n_2 el número de trabajos a realizar en los modos *normal* y *restringido* respectivamente. Para evitar el riesgo de bloqueo, no se debe utilizar una combinación con *N* tareas (donde $N = \max(n_1, n_2)$), en la que varias tareas ejecuten un trabajo correspondiente al modo normal u otro correspondiente al modo degradado dependiendo del estado global del sistema (ver listado 4.7).

En su lugar, será necesario utilizar *M* tareas (siendo $M = n_1 + n_2$), donde las n_2 de mayor prioridad realizaran un trabajo correspondiente al modo restringido, y las restantes contendrán los trabajos necesarios en el modo normal (ver figura 4.6).

Listado 4.7: Una misma tarea incluye dos modos de funcionamiento

```
task Ejemplo is
begin
   loop
      if Estado_Global.Normal then
         -- Realizar_Trabajo_Util_Modo_Normal;
      elsif Estado_Global.Restringido then
         -- Realizar_Trabajo_Util_Modo_Restringido;
      else
         -- Otros modos...
      end if;
      delay until Proxima_Activacion;
   end loop;
end Ejemplo;
```

Teniendo en cuenta que las tareas de la banda asignada al modo restringido no ejecutan ningún trabajo mientras que no se detecte ninguna anomalía y el sistema continúe operando en modo normal, no tiene sentido que sean activadas continuamente siguiendo los mismos parámetros que en el modo restringido, por tanto, para evitar un número excesivo de cambios de contexto inútiles que conllevarían una sobrecarga muy alta, cuando el sistema opere en modo normal las tareas pertenecientes a la banda de prioridad superior permanecerán suspendidas en una entrada protegida (ver listado 4.8), de esta forma su interferencia sobre la planificación del modo normal es nula.

Listado 4.8: Trabajos pertenecientes a modos distintos mediante tareas separadas

```
task Restringido is      -- Prioridad Alta!
begin
   loop
      Modo_Restringido.Wait;
      -- Realizar_Trabajo_Util_Modo_Restringido;
      delay until Proxima_Activacion;
   end loop;
end Restringido;

task Normal is      -- Prioridad Baja!
begin
   loop
      -- Realizar_Trabajo_Util_Modo_Normal;
      delay until Proxima_Activacion;
   end loop;
end Normal;
```

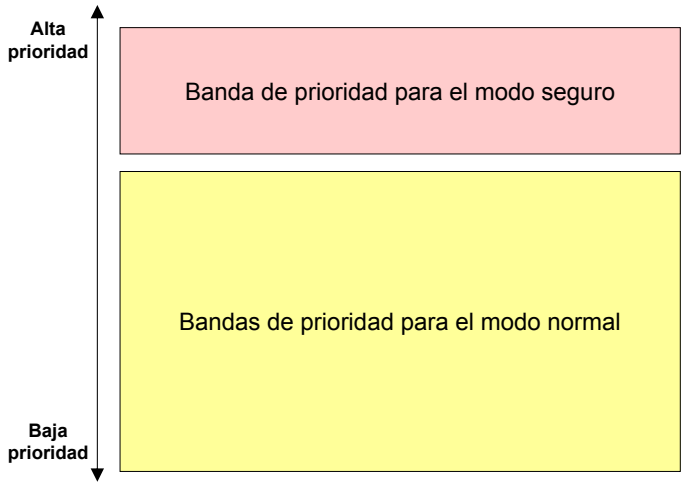

Figura 4.6: Asignación de prioridades a los diferentes modos

Reiniciar el sistema

Una última opción puede ser factible para sacar al sistema de un estado inconsistente; reiniciar el sistema. Para ello se debe tener en cuenta el tiempo necesario para llevar a cabo esta operación, ya que durante este intervalo el sistema puede quedar ingobernable, así como la posible pérdida de datos que se produciría. Dada la gran variedad de sistemas, no es posible etiquetar esta opción como válida o inválida en general. Es responsabilidad del diseñador del sistema estudiar sus características concretas y decidir si reiniciar el sistema al completo puede ser o no una opción viable.

4.7. Evaluación de opciones de diseño

Tal y como se ha descrito en los apartados anteriores, la arquitectura basada en bandas de prioridad presenta varios grados de libertad. Las diferentes alternativas con las que se puede configurar el sistema deben ser analizadas para decidir cuáles son las opciones más recomendables en cada situación. Concretamente, se debe tomar una decisión relacionada con los siguientes aspectos:

- Política de asignación de prioridades.
- Políticas de planificación utilizadas, especialmente en los planificadores locales.

- Método de tratamiento de fallos temporales.

4.7.1. Asignación de prioridades

La asignación de prioridades es una de las decisiones más importantes a la hora de diseñar un sistema de tiempo real. Como ya se comentó en el capítulo 2, probablemente el método más utilizado para asignar prioridades es DMS, ya que al ser un algoritmo óptimo y muy sencillo de implementar, conduce a los mejores resultados posibles.

Por otra parte, también se ha expuesto en el apartado correspondiente que la teoría de planificación con prioridades fijas establece que en caso de sobrecarga, aquellas tareas con un nivel de prioridad menor serán las primeras en incumplir sus plazos de respuesta. Este hecho conduce a pensar en un sistema en el que las aplicaciones estén clasificadas según su importancia (ver figura 4.7), de manera que se puedan asignar las prioridades de manera proporcional a dicho factor (a mayor importancia, mayor prioridad),con la finalidad de asegurar que si algún motivo provoca una sobrecarga temporal, serán las funciones más superfluas las primeras en incumplir sus plazos, dando de esta manera un pequeño margen a las aplicaciones más vitales para mantener el sistema en condiciones de operar aunque su funcionalidad se vea degradada.

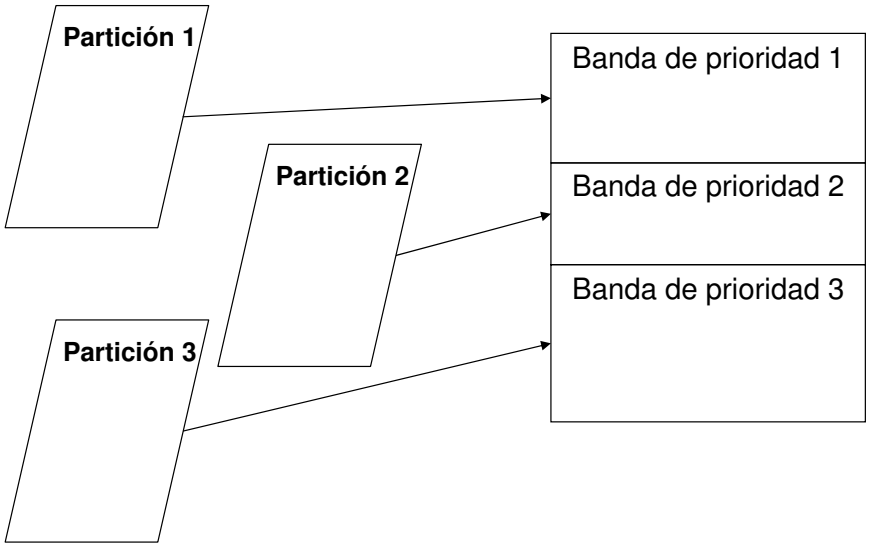

Figura 4.7: Asignación dependiente de la importancia de la aplicación

Obviamente, la importancia de una tarea no tiene ninguna correlación con sus paráme-

tros temporales (periodo, plazo de respuesta o tiempo de cómputo), por ello, un algoritmo basado en la importancia puede distar mucho de ser óptimo. En otras palabras, con un algoritmo de asignación de prioridades basado en la importancia de las tareas no se conseguirán valores de utilización del procesador tan altos como usando DMS. No obstante, para minimizar este efecto, si bien las aplicaciones están clasificadas por importancia, a las tareas que componen cada aplicación sí se le asignan prioridades utilizando DMS localmente. En la figura 4.7 se apreciaba como a cada aplicación se le asigna una *banda de prioridad*, de manera que la tarea de menor prioridad perteneciente a la aplicación Π_1 tiene una prioridad más alta que la mayor de todas aquellas pertenecientes a Π_2. Ahora bien, el mecanismo para asignar las prioridades de manera local a una aplicación es DMS, por lo que la mayor prioridad de la banda 1 es para la tarea de la aplicación Π_1 con un plazo de respuesta más corto, mientras que la menor prioridad de la banda 1 le corresponderá a la tarea perteneciente a la aplicación Π_1 con un periodo más largo.

Para cuantificar el efecto que produce este algoritmo de asignación de prioridades, se ha realizado un experimento que consiste en lo siguiente:

- Se genera un grupo de 40 tareas con parámetros semi-aleatorios (tiempo de cómputo, periodo y nivel de importancia).
- Se realiza el análisis de tiempo de respuesta y se comprueba si el conjunto es planificable o no, primero asignando las prioridades según DMS y después según la importancia de la aplicación (CMS).
- Se anota el resultado y se vuelve al punto 1.

Después de repetir este bucle 10.000 veces, se comprueba cuántos conjuntos han resultado viables con uno y otro método, lo cual permite tomar en consideración el impacto de utilizar cada opción. Para refinar las conclusiones, los parámetros no son totalmente aleatorios, sino que se ha repetido el experimento en varias ocasiones forzando alguna relación entre parámetros. Así, se ha tenido en cuenta la relación entre los periodos de las tareas y la carga de procesador que genera cada grupo.

Las figuras 4.8, 4.9 y 4.10 muestran los resultados del estudio. En el eje de abcisas se muestran los valores discretos que se han utilizado de carga del procesador (U), yendo desde el 5 % hasta el 95 % en pasos de 5 % en 5 %. El eje de ordenadas representa el número de conjuntos de tareas que han resultado viables. En los tres gráficos, la línea azul muestra el comportamiento de la asignación DMS.

La diferencia entre las figuras viene dada por los parámetros utilizados a la hora de generar los valores aleatorios de los periodos de las tareas. Así, en la figura 4.8 el periodo mayor es sólo 2 veces más largo que el menor. En la figura 4.9 este ratio se amplía hasta 5, y por último, la figura 4.10 muestra los resultados cuando el periodo más largo es un orden de magnitud mayor que el más corto.

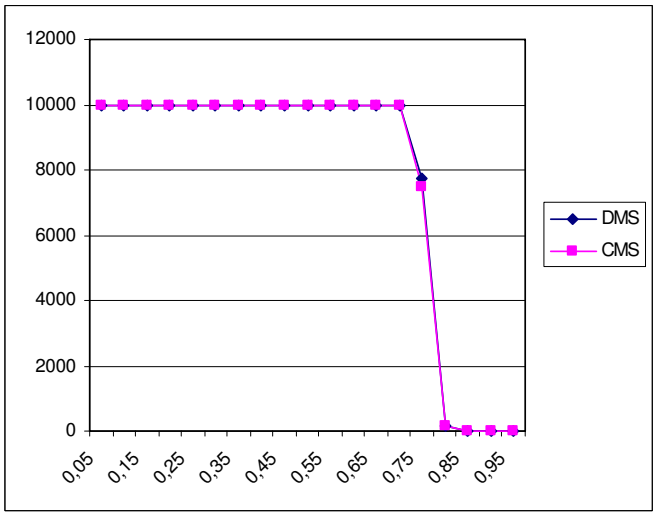

Figura 4.8: Resultados con ratio 2

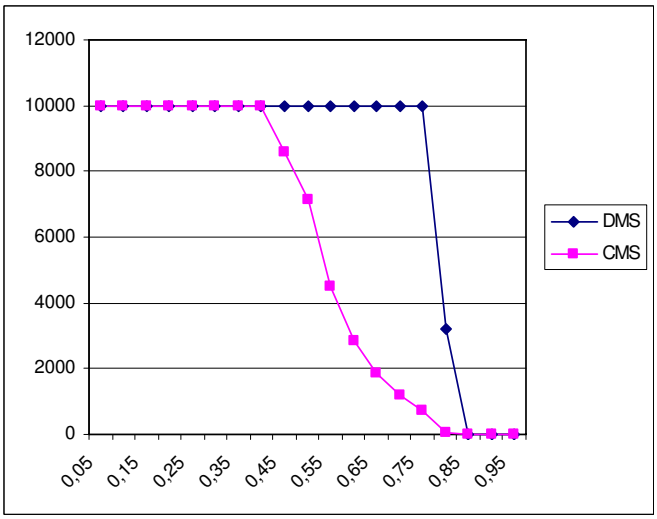

Figura 4.9: Resultados con ratio 5

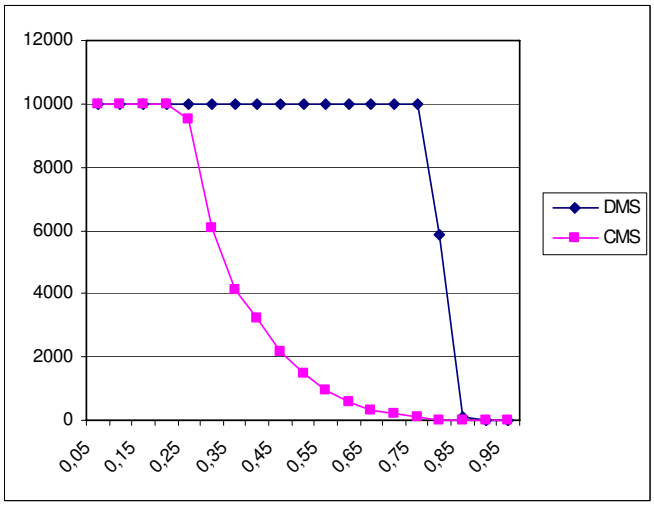

Figura 4.10: Resultados con ratio 10

La pérdida de planificabilidad viene dada por aquellas tareas con tiempos de cómputo muy largo y alta prioridad, que dejan bloqueadas a tareas con periodos cortos, no pudiendo cumplir sus plazos.

Una técnica a utilizar en caso de que un conjunto de tareas no sea viable consiste en realizar un análisis de sensibilidad, de manera que se modifiquen los parámetros de las diferentes tareas para intentar conseguir un conjunto de valores que hagan el sistema planificable. Este análisis le proporciona al diseñador del sistema más información que la simple respuesta *booleana* afirmativa o negativa que se obtiene mediante el análisis de tiempo de respuesta, ya que le permite comprobar qué parámetros son los que están empeorando la planificabilidad (Bini et al., 2006). Es más, si el análisis de sensibilidad devuelve un conjunto de valores con los que el sistema se vuelve planificable, entonces el diseñador tiene la posibilidad de evaluar si con dichos valores la funcionalidad del sistema es conservada, en cuyo caso la alternativa ofrecida es válida y se consigue transformar en viable un conjunto de tareas inviable con relativamente poco esfuerzo del diseñador del sistema (Panunzio, 2006; Panunzio and Vardanega, 2007).

Con objeto de mitigar los efectos reseñados en las figuras, es razonable pensar en una ampliación del análisis de sensibilidad, de manera que aquellas tareas con periodos y tiempos de cómputo largos (que generalmente realizan tareas de background), sean desplazadas a una banda de prioridad inferior (ver figura 4.11), con lo que se consigue minimizar su interferencia y aumentar la carga del procesador. Este modelo de asignación mixto puede ser interpretado como un compromiso entre las dos opciones comentadas

anteriormente (DMS y CMS), en la que la primera maximiza la eficiencia mientras que la segunda aumenta la seguridad.

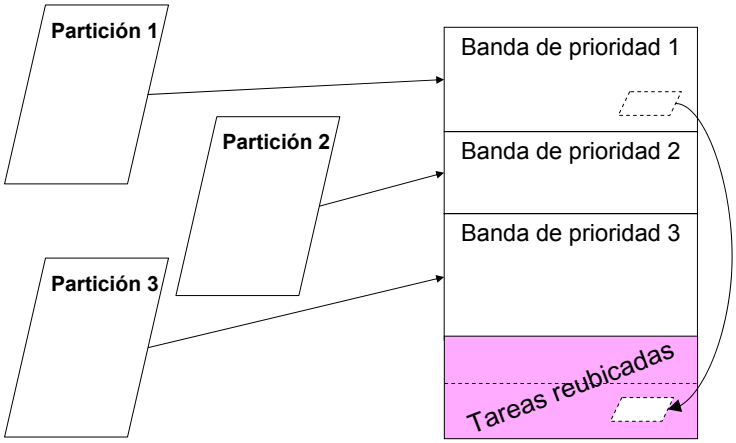

Figura 4.11: Modelo de asignación mixto

Recomendación

La arquitectura basada en bandas de prioridad cuenta con varias barreras de contención que evitan que un mal funcionamiento temporal se propague y afecte a diversas aplicaciones más allá de la que ha provocado el fallo. En primer lugar, se cuenta con el análisis estático, que ya permite predecir si un conjunto de tareas es viable o no. Posteriormente, ya en tiempo de ejecución, se cuenta con todos los métodos expuestos en este mismo capítulo para garantizar que si se produce un fallo, éste es detectado inmediatamente y tratado para evitar su propagación y salvaguardar la integridad del sistema.

La asignación de prioridades en función de la importancia de las aplicaciones puede verse como otra barrera de contención de fallos que aumenta aún más el nivel de seguridad de la arquitectura basada en bandas de prioridad. Sin embargo, las medidas enumeradas en el párrafo anterior ya proporcionan un nivel de seguridad muy alto, por lo que no se debe despreciar en absoluto la asignación de prioridades utilizando el algoritmo DMS.

Por consiguiente, después de analizar las consecuencias de utilizar las diferentes políticas de asignación de prioridades, la recomendación es comenzar utilizando la asignación basada en la criticidad de la aplicación. Si el resultado del análisis estático es positivo utilizar directamente esta técnica. En caso contrario, realizar un análisis de sensibilidad

para modificar la prioridad de aquellas tareas que más interferencia producen, llegando al extremo de utilizar la asignación DMS en caso de que sea la única asignación de prioridades viable. Debe tenerse siempre en mente que DMS no es un método inseguro ya que el resto de los elementos de la arquitectura proporciona un nivel de protección suficientemente alto como para ejecutar diferentes aplicaciones en el mismo nodo con total garantía de aislamiento temporal entre ellas.

4.7.2. Políticas de planificación

Según la descripción dada de la arquitectura de bandas de prioridad, el planificador clásico del modelo Ravenscar, basado en prioridades fijas, viene a desempeñar el papel de planificador global, por lo que en este nivel de jerarquía no existe libertad; se emplea la política de planificación de prioridades fijas. Sin embargo, gracias a la introducción de las bandas de prioridad propiamente dichas, sí que es posible tener planificadores locales configurados para usar diferentes protocolos. En esta tesis se ha decidido analizar el comportamiento de dos de ellos, prioridades fijas y EDF. El primer método ha sido seleccionado por ser el más empleado durante las últimas décadas en los sistemas de tiempo real. Por otra parte, tal y como se mostraba en el apartado 2.2.3, la teoría de EDF muestra que se puede llegar a alcanzar el 100 % de uso del procesador, por lo que en los últimos años esta política de planificación ha despertado un gran interés y hay numerosos grupos de investigación trabajando para reducir sus desventajas y explotar su eficiencia teórica (Buttazzo, 2005).

Por este motivo, viendo los resultados obtenidos en el apartado anterior al utilizar la asignación de prioridades basada en la importancia de la aplicación, parece conveniente estudiar si se puede mejorar el resultado cambiando la configuración de los planificadores locales para que utilicen EDF. Con esta configuración volveríamos a tener bandas de prioridad disjuntas ordenadas por la valía de la aplicación tal y como se mostraba en la figura 4.7, pero la organización interna a cada aplicación sería distinta ya que se utilizarían prioridades dinámicas en lugar de estáticas.

Para comprobar el resultado de esta configuración se ha realizado un experimento similar al comentado en el apartado anterior. Tras definir un conjunto de tareas con valores semi-aleatorios, se comprobaba su viabilidad tras asignar prioridades fijas siguiendo los esquemas DMS y CMS, así como asignando prioridades dinámicas con el protocolo EDF (respetando las particiones). La figura 4.12 muestra que la diferencia entre los resultados obtenidos al utilizar EDF en los planificadores locales es prácticamente inexistente.

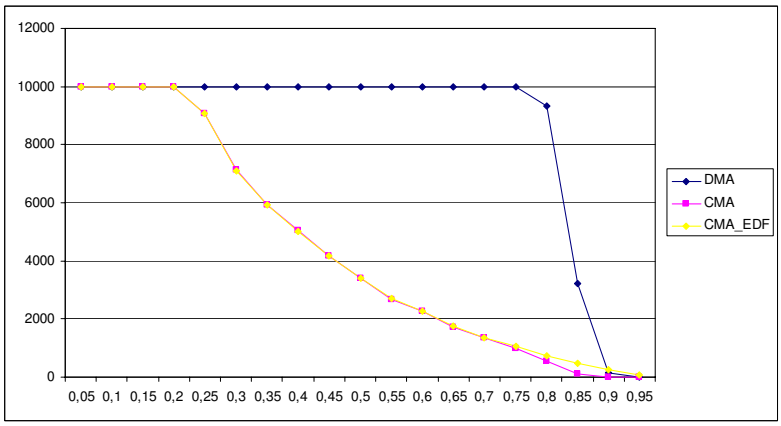

Figura 4.12: Comparación de resultados obtenidos utilizando EDF

Puesto que los resultados no muestran una mejora evidente en la utilización de EDF, unido al hecho de que implementar un núcleo que soporte dicha política de planificación no es en absoluto trivial, esta solución se ha descartado por completo, por lo que se recomienda la utilización de prioridades fijas en los planificadores tanto a nivel global como a nivel local.

4.7.3. Incumplimientos temporales

A lo largo de las secciones 4.5 y 4.6 se han explicado respectivamente diferentes maneras de monitorizar el comportamiento en tiempo de ejecución de tareas o aplicaciones y diversos mecanismos a poner en marcha en caso de que se detecte un comportamiento que no se ajusta a lo previsto mediante el análisis estático. El objetivo de este apartado es proporcionar una guía para su uso, tratando de obtener el máximo beneficio de todo este conjunto de herramientas puestas a disposición del diseñador del sistema.

Opciones de detección

Para detectar incumplimientos temporales se ha detallado el funcionamiento de cinco mecanismos:

- Detección simple
- Detección inmediata

- Cuotas colectivas
- Vigilancia del plazo de respuesta
- Cumplimiento del periodo mínimo entre activaciones esporádicas

Salvo el mecanismo de las cuotas colectivas, que ha sido desestimado, las cuatro técnicas restantes no son excluyentes, es decir, no hay necesidad de elegir una y descartar las demás, ya que es posible utilizar una combinación de varias de ellas; de hecho, es posible utilizar las cuatro simultáneamente. Sin embargo, esta última no es una opción muy eficiente.

La detección simple es la opción mas desaconsejable para monitorizar el comportamiento en tiempo de ejecución ya que no es posible predecir a priori cuándo se va a detectar el fallo, es decir, hay una latencia entre que se produce el incumplimiento y éste se detecta cuya duración es totalmente impredecible. Por este motivo, el mecanismo de detección simple puede ser adecuado en prototipos para hacer estadísticas del tiempo de procesador consumido por un cierto código, pero no lo es para detectar fallos temporales en un sistema real en el que se deba poner remedio en el instante en que se producen.

La técnica denominada *detección inmediata* proporciona una información muy completa ya que permite identificar no sólo la aplicación defectuosa sino la tarea concreta que monopoliza el uso del procesador. Además la detección se produce en el preciso instante en el que una tarea excede su WCET, por lo que el sistema está a tiempo de tomar medidas para evitar que otras tareas pierdan su plazo de respuesta. La recomendación final es utilizar siempre el mecanismo de detección inmediata.

La vigilancia de los plazos de respuesta mediante *timing events* es la última barrera de seguridad que se puede utilizar. Ésta es una alarma que salta justo cuando se cumple la *deadline* de una tarea sin que ésta haya finalizado su trabajo por lo que su uso es recomendable en todos los casos en que la tarea sea crítica y haya que tomar medidas extraordinarias de manera inmediata. Si por el contrario, el diseñador estima que la labor de ciertas tareas son poco importantes, y no es necesario tomar medidas en caso de que se incumplan sus plazos de respuesta, entonces no tiene sentido utilizar este mecanismo.

Por último, el mecanismo que garantiza que el cumplimiento del periodo mínimo entre aplicaciones esporádicas debe ser utilizado en todos los casos, ya que la sobrecarga añadida es prácticamente nula mientras que una activación incontrolada de una tarea esporádica tiene consecuencias totalmente imprevisibles para la planificabilidad del sistema.

Opciones de tratamiento

Una vez detectado el incumplimiento temporal es necesario tomar medidas para evitar que el sistema entre en un estado inconsistente que podría desembocar en una catástrofe. A lo largo del capítulo se han explicado diferentes tipos de actuaciones para tratar la situación:

- Notificar el fallo
- Utilizar el algoritmo de la segunda oportunidad
- Cambio de modo

Parece lógico que lo primero que debe hacerse en caso de detectar un fallo sea notificarlo, de esta manera, el usuario del sistema tiene información precisa para tomar las decisiones oportunas. Además, si el fallo detectado es que una tarea ha consumido su tiempo de procesador debe ponerse en marcha el algoritmo de la segunda oportunidad. Este mecanismo tiene un coste muy reducido y, sin embargo, puede ser suficiente para tratar una sobrecarga si ésta es puntual y corta. Si esta medida tiene éxito, el sistema puede continuar operando en modo normal sin necesidad de utilizar otras técnicas más complejas. Si, por el contrario, el algoritmo no tiene éxito, o bien la alarma ha sido disparada por el mecanismo que vigila el cumplimiento del plazo de respuesta, ya no ha lugar a seguir aumentando la cuota de la tarea o aplicación defectuosa ya que se estaría poniendo en peligro la integridad del resto del sistema. Es el momento de que el usuario del sistema se decida por una de las opciones etiquetadas como *cambio de modo* que incluyen la parada segura, el cambio de estado operacional a un modo restringido o el reinicio del sistema.

Es difícil dar una recomendación genérica para utilizar una opción u otra, ya que todas ellas son muy dependientes del tipo de sistema en el que se pretendan utilizar. Podríamos decir que con aplicaciones de muy baja criticidad, que no interfieran con las demás del sistema (estén asignadas a la banda de prioridad más baja) es posible incluso no tomar ninguna medida extraordinaria; notificar el fallo y permitir que se siga ejecutando. Al fin y al cabo, la banda de prioridad más baja se puede considerar de *background* ya que sólo tomará el control del procesador cuando en las demás bandas de prioridad no haya ninguna tarea lista para ejecutar. En los demás casos, queda a elección del diseñador qué opción tomar, siendo quizás la parada segura y posterior reinicio del sistema la alternativa más recomendable en términos de seguridad siempre y cuando la parada sea una opción válida (por ejemplo, no se puede hacer una parada segura de todos los subsistemas de un avión).

4.8. Aislamiento espacial

Para dar soporte a los requisitos de la máquina virtual números 1 y 2, "soporte para diferentes niveles de criticidad" y "contención de fallos" respectivamente, además de proporcionar aislamiento temporal entre las aplicaciones, también se debe hacer énfasis en el aislamiento espacial con objeto de evitar que una aplicación acceda a una zona de memoria que no le corresponde. El objetivo de esta tesis comprende el estudio e implementación del aislamiento temporal, sin embargo, la implementación del aislamiento espacial queda fuera de su ámbito. Esta sección apunta diversos métodos que permiten proporcionar aislamiento espacial y que pueden ser investigados en el futuro, complementando los estudios realizados sobre el aislamiento temporal para poder proporcionar un aislamiento entre particiones completo (Urueña and Zamorano, 2007).

Existen tres técnicas básicas que permiten proporcionar aislamiento espacial entre aplicaciones:

- Análisis estático de flujo de información
- Comprobaciones en tiempo de ejecución
- Protección *hardware* contra escritura

4.8.1. Análisis estático de flujo de información

Un análisis estático acerca de los flujos de información realizado sobre el código fuente puede garantizar que una aplicación no escribirá en una zona de memoria perteneciente a otra aplicación. Utilizando un lenguaje como *Spark* (Barnes, 2003), que permite añadir anotaciones a subprogramas y variables, se puede seguir la evolución del flujo de información para detectar potenciales fuentes de errores, por ejemplo, que un subprograma intente modificar el valor de una variable clasificada a un nivel de criticidad más alto (Amey et al., 2005).

La información que se puede obtener utilizando este método es muy valiosa, sin embargo, exige que todo el *software* sea programado utilizando Spark. Esta premisa puede ser viable para los niveles de criticidad más altos, sin embargo, carece de sentido implementar con Spark los niveles no-críticos, ya que Spark es un lenguaje muy escueto que permite muy pocas construcciones, es decir, carece de la funcionalidad suficiente para implementar los niveles de criticidad más bajos.

Por otra parte, sí que sería muy interesante investigar la mejora de la eficiencia en la ejecución de aplicaciones criticas que utilicen Spark, puesto que la garantía obtenida estáticamente permite eliminar en parte el uso de otros mecanismos *hardware* de protección, cuya sobrecarga no es despreciable.

4.8.2. Comprobaciones en tiempo de ejecución

El lenguaje Ada no sólo permite hacer comprobaciones durante la compilación del sistema, sino que también lleva a cabo en tiempo de ejecución numerosas comprobaciones que no pueden ser realizadas al compilar, por ejemplo, desreferenciar un puntero nulo o acceder fuera de los límites de un *array*.

Esta técnica se puede aplicar también para garantizar aislamiento espacial, ampliándola para comprobar que una tarea no accede a posiciones a las que no debería tener acceso, por ejemplo, verificando que un puntero no apunta fuera del rango permitido. El punto más desfavorable de este método es la necesidad de confiar en el compilador a la hora de generar estos puntos de control, el cual es un elemento demasiado complejo como para someterlo a un proceso de certificación al máximo nivel. Asimismo, es necesario investigar la sobrecarga introducida por esta clase de comprobaciones en tiempo de ejecución, ya que también podrían alcanzar valores no despreciables.

4.8.3. Protección hardware contra escritura

Los mecanismos de protección de memoria descritos hasta ahora están basados solamente en *software*, sin embargo, también se puede hacer uso de recursos *hardware* para proporcionar aislamiento espacial.

- Una unidad de gestión de memoria (MMU) permite la traducción entre direcciones y direcciones virtuales. Este componente permite que una aplicación sólo maneje un rango de direcciones virtuales (que la MMU se encarga de transformar en direcciones físicas) ocultando así aquellas direcciones a las que la aplicación no debe acceder.

- Una pareja de *registros valla* definen la dirección inicial y final de un rango. Su utilización permite prohibir escrituras, bien dentro del rango delimitado por los registros, bien fuera de él.

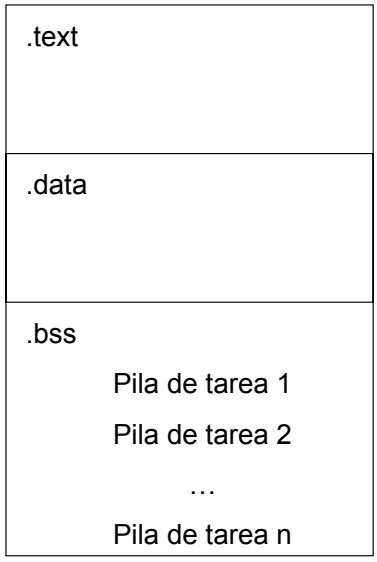

Figura 4.13: Mapa de memoria actual en GNATforLEON

La plataforma LEON2 hacia la que está orientada la máquina virtual ofrece un pobre soporte en lo que a elementos *hardware* se refiere, ya que solamente cuenta con dos parejas de registros valla. No existe ninguna clase de MMU por lo que no hay posibilidad de traducciones de memoria vía *hardware*. Todas las aplicaciones comparten un mismo espacio de memoria.

En la versión actual de GNATforLEON todo el código se enlaza estáticamente en un único archivo binario. La pilas de las tareas se crean en la misma zona de memoria durante la inicialización. La figura 4.13 muestra el mapa de memoria. Los registros valla no se utilizan puesto que todas las variables globales se encuentran en la misma sección (*.data* o *.bss*) independientemente de su nivel de criticidad. Además, el modelo *proxy* que utiliza GNAT para implementar los objetos protegidos, según el cual, una tarea que ejecuta un procedimiento protegido puede tener que escribir en la pila de una tarea encolada en una entrada protegida, también impide el uso natural de los registros valla.

La figura 4.14 muestra un mapa de memoria alternativo. En él, código, datos y pila de tarea están claramente separados cuando pertenecen a particiones distintas. En el futuro se puede plantear una modificación de GNATforLEON para generar un mapa de memoria semejante al de la figura, ya que este esquema sí que permite la protección *hardware*. Incluso si las próximas versiones de la plataforma LEON incluye un mayor número de

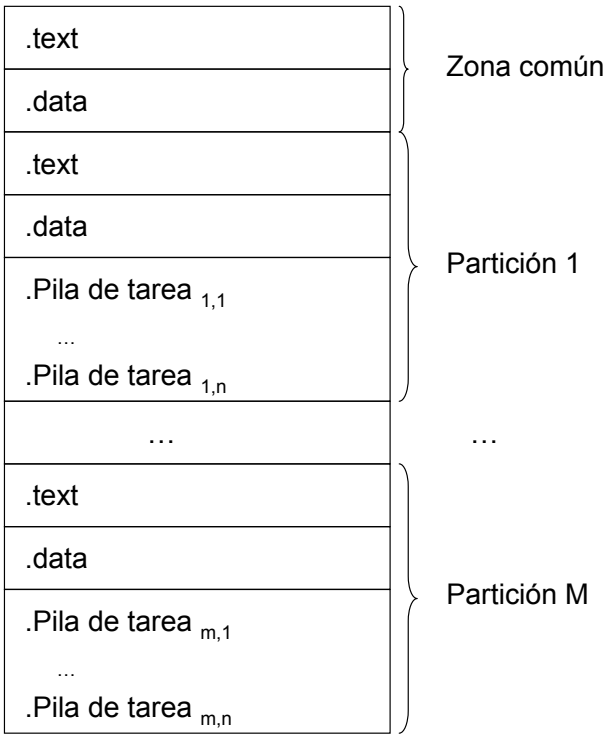

Figura 4.14: Posible mapa de memoria para GNATforLEON en el futuro

registros valla, se podría mejorar la granularidad de la protección.

Capítulo 5

MODELO DE COMPONENTES

5.1. Un nuevo proceso de desarrollo

El crecimiento constante de los costes asociados a un proyecto de ingeniería de *software* supone un importante freno en la productividad. El desarrollo de sistemas de tiempo real empotrados no es ajeno a este problema, por lo que resulta fundamental ofrecer soluciones orientadas a las distintas causas que lo generan.

El capítulo 4 se ha centrado en mostrar una nueva arquitectura que aumenta la flexibilidad y, por tanto, la eficiencia, garantizando además un nivel de seguridad suficiente como para ser utilizada en sistemas de alta integridad. Obviamente el diseño de una arquitectura eficiente puede mejorar considerablemente el rendimiento del sistema; si además es configurable se gana en reutilización, ya que se podrían adaptar sus características a los requisitos concretos de cada proyecto, lo cual evita tener que empezar desde cero cada vez y por consiguiente se ahorran costes.

Sin embargo, mejorar el diseño arquitectónico sólo resuelve parte del problema. Actualmente se calcula que el coste de las fases de integración y pruebas de un proyecto de ingeniería de *software* puede llegar a ascender hasta el 50 % del total. Ello es debido a errores cometidos a lo largo de las diferentes fases del proyecto. Se pueden identificar dos fuentes principales que conducen a estos errores:

- Unas especificaciones poco claras o ambiguas hacen que la implementación acabe reflejando un concepto distinto de la idea original.
- El factor humano siempre introduce errores en la fase de implementación.

Como consecuencia, la fase de integración se convierte en un proceso largo y tedioso en el que se acaba invirtiendo una gran cantidad de esfuerzo. Asimismo, es necesario llevar a cabo un programa de pruebas exhaustivo, necesario para detectar y corregir el mayor número de errores que sea posible. Todo ello, se refleja finalmente en los costes del proyecto.

Por tanto, si se desea actuar eficazmente contra la escalada de costes es necesario atacar el problema desde todos los frentes. Con este objetivo, además de las medidas centradas en el diseño de una nueva arquitectura propuestas hasta este momento, este capítulo detalla una nueva metodología que persigue un doble objetivo:

- Facilitar el trabajo del diseñador del sistema, ocultándole en la medida de lo posible los detalles de bajo nivel, y ofreciéndole una serie de componentes combinables para formar sistemas correctos por construcción.
- Establecer un proceso de generación automática de código que minimice los errores introducidos por el factor humano.

5.2. Diseño basado en contenedores

En el marco del proyecto ASSERT se ha definido un nuevo método de desarrollo que fomenta las prácticas expuestas anteriormente. Por un lado, se hace hincapié en separar las propiedades funcionales y no funcionales del sistema, de manera que el diseñador puede centrarse en la parte funcional y no debe tener en mente los detalles no funcionales de bajo nivel en las etapas tempranas del desarrollo, lo cual facilita sensiblemente su labor. Esto tiene un impacto importante en el coste del proyecto, ya que el hecho de ser una labor más sencilla tiene dos implicaciones:

- Se necesita menos tiempo para completar la fase de diseño, por lo que se produce un ahorro *directo*.
- Se cometen menos errores, que de otra forma serían trasladados a fases futuras, alargando la integración y las pruebas del sistema, por lo que también se produce un ahorro *indirecto*.

La figura 5.1 muestra el proceso completo. En ella se ilustra el modo en el que durante el modelado se pueden expresar las propiedades no-funcionales de interés y preservarlas

a lo largo del proceso de desarrollo completo, desde la especificación hasta el despliegue sobre un entorno de ejecución. Para que esto sea posible, el modelo del sistema construido durante la fase de diseño debe incluir cuatro aspectos complementarios:

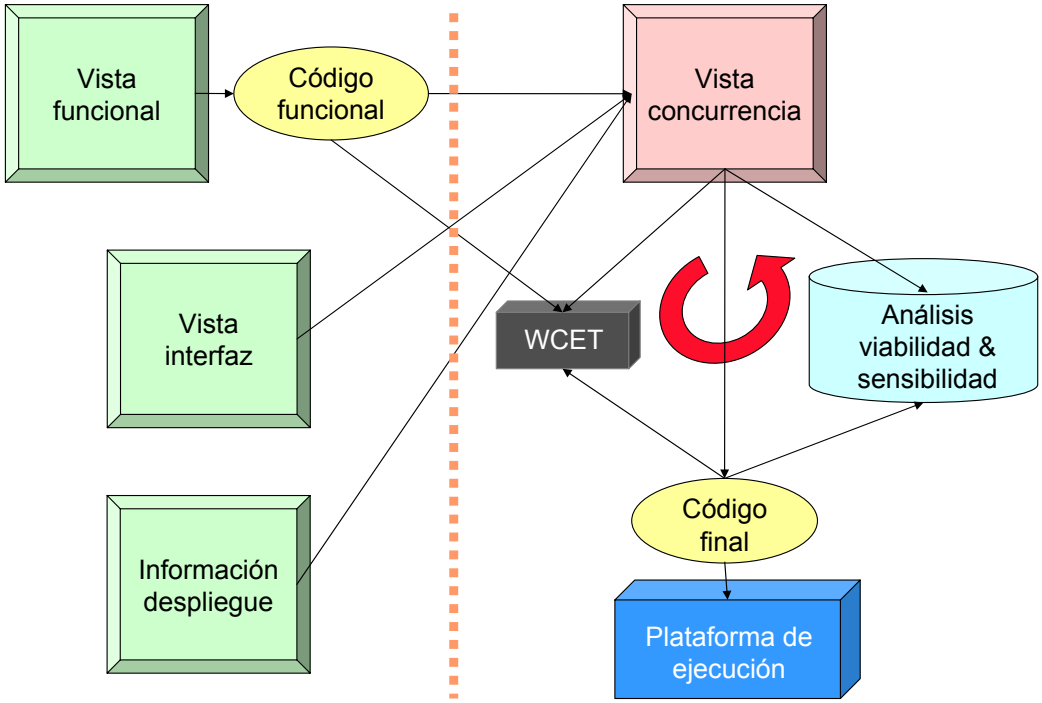

Figura 5.1: Proceso de desarrollo en ASSERT

- Interfaces: especifica las interfaces de los componentes del sistema así como las relaciones entre ellas.
- Funcionalidad: en cada componente se describe su funcionalidad interna mediante el formalismo adecuado, respetando siempre las restricciones impuestas por el modelo computacional elegido para el sistema (en nuestro caso particular, el modelo Ravenscar).
- Despliegue: detalla las características de la plataforma de ejecución y la manera de desplegar el *software* sobre el *hardware*, incluyendo, por ejemplo, la definición de las particiones a realizar.
- Concurrencia: determina los parámetros fundamentales de las hebras de control que existen en el sistema, incluyendo su sincronización y los protocolos utilizados,

cumpliendo siempre los requisitos necesarios para poder llevar a cabo un análisis estático.

A partir del modelo funcional se genera su código fuente. Éste, junto con la información necesaria para el despliegue sobre una plataforma *hardware* concreta y los datos de concurrencia básicos, se integra dentro de unos bloques que conforman la parte no-funcional del sistema. Todo ello, pasa por un proceso para determinar el tiempo de ejecución en el peor caso posible y por un ciclo de análisis de sensibilidad y viabilidad. Tal y como se apuntaba en el capítulo anterior, el análisis de viabilidad se corresponde con el análisis estático explicado en la sección 4.4 que permite conocer si conjunto final es planificable o no. En caso de que la respuesta sea negativa, el análisis de sensibilidad permite modificar los parámetros de tiempo real utilizados con objeto de encontrar un conjunto de valores que hagan el sistema viable. Estos nuevos valores son mostrados al diseñador del sistema que debe dar el visto bueno final confirmando que los nuevos parámetros (por ejemplo, un periodo más largo de la tarea $\tau_{i,j}$), entran dentro de un rango suficientemente razonable como para que el sistema continúe cumpliendo sus requisitos funcionales. Después de esta fase iterativa, se genera el código fuente final que se va a desplegar finalmente sobre la plataforma de ejecución.

5.2.1. Conceptos clave

Para dar soporte al procedimiento expuesto, se necesitan cuatro elementos básicos que se describen a continuación: contenedores, máquina virtual, transformaciones y propiedades.

Contenedores

El elemento básico que permite integrar de manera coherente las diferentes perspectivas representadas en la figura 5.1 es el *contenedor*. Los contenedores, mediante atributos, valores y características permiten expresar estrictamente el grado de libertad especificado en el modelo de proceso elegido para desarrollar el sistema. Una vez más, en el contexto de esta tesis nos estamos refiriendo al modelo Ravenscar.

Para facilitar el trabajo, tanto el nuestro a la hora de especificar los contenedores, como el del diseñador del sistema a la hora de utilizarlos, los contenedores se han dividido en dos categorías.

- Contenedores de aplicación (CAP): pensados para incluir la perspectiva funcional y la de interfaz, ocultando los detalles de bajo nivel referentes a la concurrencia.
- Contenedores de máquina virtual (CMV): definidos como entidades que existen en tiempo de ejecución y que contienen una serie de propiedades garantizadas por la *máquina virtual*.

Máquina virtual

La máquina virtual es una plataforma de ejecución, entendiendo como tal, una abstracción de *hardware* y *software* capaz de planificar y ejecutar aplicaciones multi-hebra distribuidas. Su arquitectura ya ha sido presentada en el apartado 4.2, por lo que esta sección se centra en las propiedades que permiten que sea uno de los actores principales del proceso de desarrollo.

Si bien hasta ahora se había presentado como su principal activo la flexibilidad que le permite planificar las tareas de manera eficiente, a su vez, la plataforma debe ser inflexible cuando se trata de hacer cumplir estrictamente las restricciones enumeradas en el modelo de proceso, ejecutando y vigilando el comportamiento en tiempo de ejecución de los contenedores que pueden operar sobre ella. Las características principales de la máquina virtual son las siguientes:

- Implementa un modelo computacional concurrente que se puede someter a un análisis estático. Este modelo permite la comunicación entre tareas mediante elementos intermedios encargados de la sincronización que no incurren en indeterminismo.
- Es un entorno de ejecución que sólo acepta y da soporte a las entidades *legales* definidas por la semántica del modelo Ravenscar. Es decir, los contenedores de máquina virtual.
- Está ligada a un sistema de compilación que sólo produce código para las entidades *legales* y rechaza cualquier construcción fuera del modelo definido. Además, cuenta con una serie de mecanismos que llevan a cabo en tiempo de ejecución aquellas comprobaciones que no es posible realizar en tiempo de compilación.
- Aporta servicios en tiempo de ejecución que permiten que los contenedores conserven sus propiedades, básicamente:
  - Medir de forma precisa del tiempo de ejecución real consumido por cada tarea.

— Asignar una cuota de tiempo de ejecución a cada tarea, rellenándola según lo estimado en el modelo y disparando una alarma en caso de que una tarea agote sus recursos.

— Agrupar las tareas, asignando cuotas colectivas de tiempo de ejecución, contabilizando su consumo y reposición de manera análoga a como se hace con tareas individuales.

— Forzar el cumplimiento del periodo mínimo entre dos activaciones consecutivas de las tareas esporádicas.

— Erigir fronteras que permitan delimitar regiones de contención de fallos para evitar la propagación de estos de unas tareas a otras.

— Lograr transparencia en la distribución y comunicación entre tareas.

Transformaciones

La intención de simplificar la tarea del diseñador de sistemas mediante el modelo de proceso que se está describiendo se traduce en que aquel sólo trabaja directamente con los contenedores de aplicación, o lo que es lo mismo, sólo tiene en cuenta la perspectiva funcional y la de interfaz, así como información sobre el despliegue.

Sin embargo, al contrario que los contenedores de máquina virtual, los contenedores de aplicación no son entidades ejecutables directamente por la máquina virtual. Por tanto, es necesario llevar a cabo lo que denominamos una *transformación vertical*, que convierta los contenedores de aplicación en una serie de contenedores de máquina virtual, todo ello de manera fiable, es decir, conservando sus propiedades, sin ninguna distorsión semántica, y forzando el cumplimiento de las propiedades y restricciones impuestas por el modelo de proceso.

Propiedades

Dentro del modelo propuesto, las propiedades funcionales tienen cabida tanto en la vista funcional como en la de interfaz. El diseñador trabaja sobre ambas perspectivas. En la primera, el diseñador especifica las acciones llevadas a cabo por cada método individualmente. En la segunda, el diseñador especifica cuáles son las interfaces de los componentes (*i.e.* qué características, atributos y métodos proveen) y cómo interactúan entre sí.

Propiedades no-funcionales tales como los requisitos de tiempo real estrictos referentes a la invocación y ejecución de tareas (periodo, *deadline*...), se especifican en la vista de interfaz para que posteriormente, mediante la transformación vertical, se vean reflejadas en la vista de concurrencia.

5.2.2. Modelo del sistema

Desde el punto de vista del diseñador, el modelo del sistema está compuesto por una serie de contenedores de aplicación (CAP) interconectados entre sí. Posteriormente, una transformación vertical automatizada se encarga de convertir este modelo en otro semánticamente equivalente, pero formado ya solamente por contenedores de máquina virtual, los cuales, recordemos que son las únicas entidades aceptadas por la plataforma de ejecución.

No puede existir una jerarquía de contenedores de aplicación, es decir, un CAP no puede tener en su interior otro CAP. En otras palabras, el nivel de CAP es el máximo hasta el que puede descender el diseñador del sistema. A partir de ahí, todas las transformaciones están automatizadas, por lo que el diseñador no puede intervenir en el modelo a nivel de máquina virtual. La justificación para esta restricción viene dada precisamente por el propósito de este proyecto de automatizar en la medida de lo posible el proceso de diseño, preservando las propiedades a lo largo de todo él y evitando alteraciones introducidas por el factor humano.

Modelo computacional

El modelo computacional impuesto por este proceso coincide casi totalmente con el modelo Ravenscar, ampliamente comentado a lo largo de esta memoria. El modelo supone que se ejecuta un conjunto estático de tareas (creadas en tiempo de elaboración y que no tienen fin), concurrentes, planificadas mediante prioridades fijas, que se pueden intercomunicar mediante mecanismos de sincronización tipo monitor y con accesos a regiones críticas regladas por protocolos basados en el techo de prioridad para evitar el fenómeno de la inversión (recordar apartado 2.2.4). Además, las tareas tienen un único punto de suspensión, por lo que aparte de ese único punto, el cuerpo de una tarea no puede contener acciones potencialmente bloqueantes (instrucciones *delay*, llamadas a entradas protegidas, etcétera). La diferencia entre ambos modelos computacionales es que se ha decidido aceptar el uso de los temporizadores de tiempo de ejecución.

Garantías en tiempo de ejecución

Las tareas llevan a cabo un cierto trabajo, bien con un periodo fijo, marcado por el reloj del sistema (tarea periódica), o bien esporádicamente, provocadas por un evento externo, siempre y cuando se respete un periodo mínimo entre dos activaciones consecutivas (tarea esporádica). En cualquier caso, el tiempo de cómputo necesario para llevar a cabo dicho trabajo se supone finito y especificado.

La máquina virtual garantiza que el comportamiento en tiempo de ejecución de las tareas respeta su especificación, de manera que una tarea esporádica no viole su periodo mínimo entre activaciones consecutivas, y una tarea periódica cumpla siempre con su periodo. En ambos casos se detectan los excesos de consumo de procesador y se toman las medidas adecuadas.

5.3. Agrupando las diferentes perspectivas

A lo largo del capítulo se ha afirmado que para diseñar un sistema se han de combinar cuatro aspectos o perspectivas complementarias: interfaz, funcional, concurrencia y despliegue, donde cada uno de estos aspectos contiene un tipo de información específico. Las figuras 5.2 y 5.3 representan cómo se combinan los aspectos mencionados para dar lugar al sistema final. La primera de ellas hace énfasis en el interfaz y posterior despliegue sobre el *hardware*, mientras que la figura 5.3 representa la ampliación de un CAP para hacer énfasis en el contenido funcional y en los aspectos relacionados con la concurrencia.

En la figura 5.2 se distinguen dos contenedores a nivel de aplicación. Cada uno de estos contenedores presenta al exterior dos interfaces, IR e IP, cuyo significado es *Interfaz Requerida* e *Interfaz Provista* respectivamente. De esta manera, cada contenedor a nivel de aplicación viene definido tanto por los servicios que es capaz de proporcionar al exterior (es decir, lo que este componente puede hacer por los demás), así como por aquellos servicios que necesita obtener del exterior.

El diseñador del sistema puede combinar el número de CAP que estime oportuno, estableciendo relaciones entre las diferentes interfaces provistas y requeridas para llevar a cabo su objetivo. Por otra parte, gracias a las perspectivas de implementación y despliegue, se puede determinar el número de particiones que se desean en un sistema, decidir qué aplicaciones formarán parte de una partición, así como definir en qué nodo del sistema estarán físicamente localizadas cada una de estas particiones.

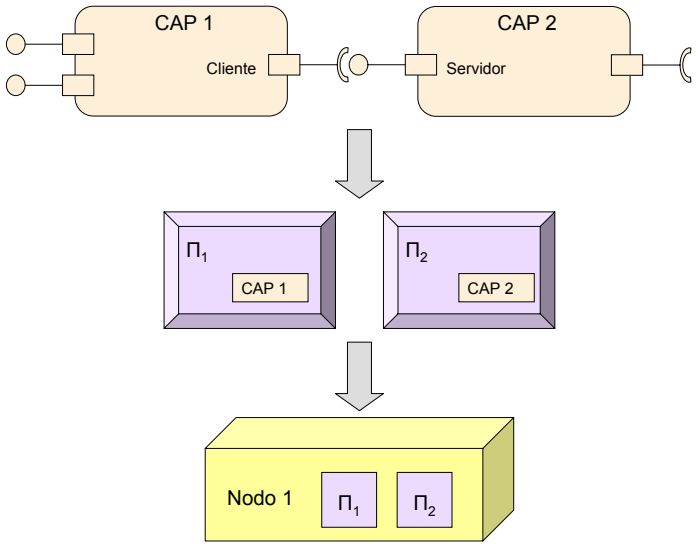

Figura 5.2: Despliegue del software sobre el hardware

La figura 5.3 muestra los aspectos complementarios, es decir, funcionalidad y concurrencia. Para facilitar la comprensión la figura muestra un caso sencillo, en el que un contenedor a nivel de aplicación se transforma directamente en un sólo contenedor de máquina virtual (en el caso general, un solo CAP puede generar varios CMV). Dicha figura muestra el CMV empotrado dentro del CAP, por lo que resulta sencillo comprender cómo se realiza la transformación.

El diseñador del sistema sólo tiene que preocuparse de generar el código funcional necesario para cada CAP. No tiene acceso a aspectos relacionados con la concurrencia más allá de especificar el periodo con el que se ha de ejecutar dicho código y su plazo de respuesta. Es responsabilidad del proceso de generación automática de código encargarse de asignar los parámetros adecuados para cumplir con los requisitos no-funcionales. Para ello, el código funcional junto con sus correspondientes atributos son mapeados sobre un contenedor a nivel de máquina virtual como el mostrado en la figura 5.3, en el que se pueden distinguir tres módulos internos:

- Tarea: hebra de control que permite la ejecución del servicio invocado a través del correspondiente interfaz provisto.

- Módulo funcional: contiene el código funcional especificado por el diseñador para proveer un determinado servicio.

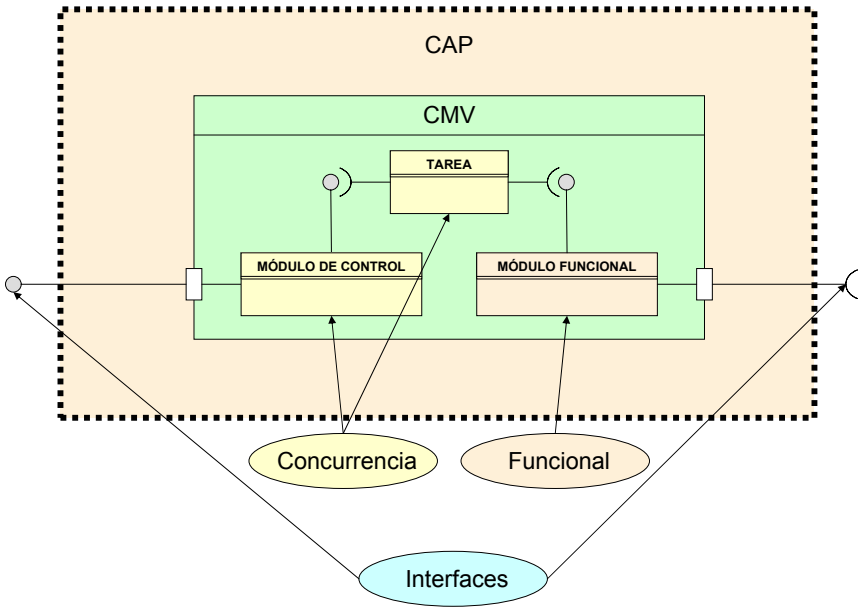

Figura 5.3: Perspectiva funcional y concurrencia

- Módulo de control: es el encargado de garantizar el correcto funcionamiento del contenedor en términos de sincronización.

En resumen, el diseñador del sistema debe crear un número de CAP, especificando sus interfaces y las comunicaciones existentes entre ellos. Asimismo, es su responsabilidad generar el código funcional correspondiente a cada CAP. De manera independiente, es necesario definir el número de particiones que se desea tener en el sistema, identificar a cuál de ellas pertenece cada CAP, así como especificar sobre qué nodo físico serán desplegadas. Todo ello teniendo en cuenta los siguientes puntos:

- Un CAP está integrado dentro de una única partición. Con objeto de proporcionar un nivel de aislamiento adecuado no está permitido que un CAP forme parte de varias particiones.

- Una partición puede contener más de un CAP. En este sentido no existen restricciones ya que es responsabilidad del diseñador elegir el nivel de granularidad de las particiones que desea implementar en su sistema. No obstante, consideramos que el nivel mínimo es una partición por cada nivel de criticidad, por lo que si en un nodo han de convivir aplicaciones certificadas a distintos niveles de criticidad, éstas han de ser ligadas a particiones diferentes.

- No puede haber más de una partición con el mismo nivel de criticidad. Con objeto de sacar el máximo provecho de la arquitectura de bandas de prioridad, es necesario poder establecer una relación de orden total entre los valores de criticidad de las distintas particiones y poder así asignar la banda con prioridad más alta a la partición con mayor nivel de criticidad.

El diseño detallado de los contenedores de aplicación así como su transformación a contenedores de nivel de máquina virtual ha sido desarrollado por otros grupos de investigación (Bordin and Vardanega, 2007). Sin embargo, sí que se ha colaborado activamente en la definición de contenedores a nivel de máquina virtual, especialmente en su transformación final a código fuente. El resto de las secciones de este capítulo están enfocadas a presentar en primer lugar el catálogo de contenedores a nivel de máquina virtual y, en segundo lugar, a definir las reglas de transformación que permiten convertir dichos contenedores en código fuente ejecutable sobre la plataforma *hardware* elegida: LEON2 (Pulido, de la Puente, Hugues, Bordin and Vardanega, 2007).

5.4. Catálogo de contenedores a nivel de máquina virtual

Los contenedores a nivel de máquina virtual son entidades ejecutables por la máquina virtual, de hecho, son las únicas construcciones legales aceptadas por ésta. Su composición interna en el caso general ya ha sido reflejada en la figura 5.3, sin embargo, en esta sección se van a definir tres tipos de contenedores en los que se mostrará que, dependiendo de su misión, alguno de sus componentes internos puede ser nulo.

5.4.1. Contenedor planificable

Los contenedores planificables o concurrentes son aquellos que representan un flujo de control. Son los principales encargados de proporcionar los aspectos de la ejecución ligados a la concurrencia y por tanto garantizar que se conservan las propiedades de tiempo real asignadas a lo largo del proceso de desarrollo.

El aspecto de un contenedor planificable es muy similar al mostrado en la figura 5.3 puesto que hace uso de sus tres subcomponentes:

- La tarea, basada en un tarea Ada, es el elemento que permite al núcleo de tiempo real de la máquina virtual planificar adecuadamente la ejecución de los diferentes

servicios.

- El módulo de control de un contenedor planificable es el elemento que regula la ejecución de la tarea, ya sea de manera periódica, dirigido por el reloj de tiempo real del sistema, o esporádica, tras recibir una petición de ejecución desde otro contenedor.

- El módulo funcional está compuesto por una serie de procedimientos que implementan la interfaz provista por el contenedor. En cada activación de la tarea, el evento que realiza la petición de ejecución (ya sea periódica o esporádica) contiene un indicador que identifica el servicio que debe ser ejecutado.

5.4.2. Contenedor protegido

El contenedor protegido representa la noción de *dato* que debe ser protegido por algún tipo de mecanismo que proporcione exclusión mutua debido a que está sujeto a accesos concurrentes.

A diferencia del contenedor planificable, el contenedor protegido sólo tiene dos subcomponentes: el módulo de control y el módulo funcional. El módulo correspondiente a la tarea es nulo. De nuevo el módulo funcional es el encargado de implementar los servicios ofrecidos a través de la interfaz provista. En este caso, el módulo de control es el encargado de seguir el protocolo de acceso a recursos compartidos adecuado para garantizar la exclusión mutua y evitar inversiones de prioridad.

5.4.3. Contenedor pasivo

Los contenedores pasivos no presentan ningún tipo de semántica concurrente. Por este motivo son los más sencillos de los tres, tanto el modulo de control como la hebra son nulos, siendo el módulo funcional el único subcomponente válido para este tipo de contenedor. Una vez más, éste es el encargado de proporcionar los servicios ofrecidos a otros contenedores externos a través de su interfaz provista.

5.5. Esquemas de implementación

El último paso en el proceso de transformación vertical es la generación de código fuente Ada 2005 a partir del conjunto de contenedores de nivel de máquina virtual generados en el paso anterior del proceso. Para que el proceso sea calificado como exitoso, el código fuente final debe respetar fielmente la funcionalidad declarada al principio del proceso dentro de lo que hemos denominado *aspecto* o *perspectiva funcional*, además de ser capaz de soportar un análisis estático que permita prever el comportamiento temporal del sistema. Precisamente por este último motivo se ha insistido a lo largo de todo el proceso en ser fiel al perfil de Ravenscar.

Los siguientes apartados muestran el código fuente necesario para implementar el catálogo de contenedores de nivel de máquina virtual mostrado en la sección anterior, en los que se ha de conjugar el apartado funcional y los elementos de control mostrados con las estructuras descritas en el capítulo 4 referentes a la arquitectura de bandas de prioridad.

5.5.1. Contenedor pasivo

En esta ocasión, para ir de lo más sencillo a lo más complejo, vamos a comenzar con el contenedor pasivo. Esta clase de contenedor es realmente simple ya que no cuenta con mecanismos de control de ningún tipo. Simplemente contiene un elemento funcional que debe ser ejecutado sin ni siquiera necesidad de exclusión mutua, por este motivo, su implementación en código Ada 2005 se reduce a un simple procedimiento por cada uno de los servicios ofrecidos a través de la *interfaz provista*. El listado 5.1 muestra un sencillo ejemplo con dos procedimientos que contienen sendos parámetros en modo *in out* que representan el intercambio de datos.

Listado 5.1: Implementación de un contenedor pasivo

```
package Contenedor_Pasivo is
   -- Un procedimiento para cada servicio ofertado
   procedure Servicio_Provisto_1 (Parametro : in out Data);
   procedure Servicio_Provisto_2 (Parametro : in out Data);
   -- ...
end Contenedor_Pasivo;

package body Contenedor_Pasivo is
   procedure Servicio_Provisto_1 (Parametro : in out Data) is
   begin
      Trabajo_Util_1;
   end Servicio_Provisto_1;

   procedure Servicio_Provisto_2 (Parametro : in out Data) is
   begin
      Trabajo_Util_2;
   end Servicio_Provisto_2;

   -- ...

end Contenedor_Pasivo;
```

5.5.2. Contenedor protegido

La implementación en Ada 2005 de un contenedor protegido es sencilla gracias a los objetos protegidos, que permiten garantizar el acceso en exclusión mutua utilizando para ello el protocolo *Ceiling Locking* que evita el fenómeno de la inversión de prioridad.

El listado 5.2 muestra el código necesario para su implementación, un tipo de dato protegido en el que se declara una prioridad techo mediante la directiva *pragma Priority* o *pragma Interrupt_Priority*. Además de este listado, debe tenerse en cuenta que es necesario incluir en la compilación un archivo denominado *gnat.adc*, en el que se añaden directivas para el compilador, que bien mediante la linea

$$pragma\ Locking\_Policy\ (Ceiling\_Locking)$$

o bien mediante alguna otra directiva más genérica que incluya la anterior, como por ejemplo

$$pragma\ Profile\ (Ravenscar)$$

fuerce la utilización del protocolo de techo de prioridad en los objetos protegidos.

Dichas directivas pueden ser vistas como la implementación del módulo de control que gestiona el acceso. Los diferentes procedimientos declarados e implementados se corresponden con el módulo funcional. Por último, el único atributo necesario para completar la implementación es el *techo de prioridad* del contenedor, representado en el listado por *Mi_Techo*.

Listado 5.2: Implementación de un contenedor protegido

```
-- El archivo gnat.adc incluye la directiva:
-- pragma Locking_Policy (Ceiling_Locking);

package Contenedor_Protegido is
   protected type Dato_Protegido is
      pragma Priority (Mi_Techo)
      -- Un procedimiento para cada servicion ofertado
      procedure Servicio_Provisto_1 (Parametro : Data);
      procedure Servicio_Provisto_2 (Parametro : Data);
      -- ...
   private
      -- Zona privada
   end Dato_Protegido;
end Contenedor_Protegido;

package body Contenedor_Protegido is
   protected body Dato_Protegido is
      procedure Servicio_Provisto_1 (Parametro : Data) is
      begin
         Trabajo_Util_1;
      end Servicio_Provisto_1;

      procedure Servicio_Provisto_2 (Parametro : Data) is
      begin
         Trabajo_Util_2;
      end Servicio_Provisto_2;

      -- ...

   end Dato_Protegido;
end Contenedor_Protegido;
```

5.5.3. Contenedor planificable

Tal y como se ha visto en la sección anterior, el contenedor planificable (o concurrente) es el más complejo de todos porque sus tres subcomponentes son no nulos. La implementación de este tipo de contenedores se hace mediante una tarea Ada 2005. Para

ser coherente con el modelo computacional exigido por Ravenscar, el cuerpo de la tarea es un bucle infinito que no contiene llamadas potencialmente bloqueantes exceptuando el único punto de entrada permitido (una instrucción *delay until* en el caso de una tarea periódica o una entrada protegida si se trata de una tarea esporádica).

Existen dos tipos de contenedores concurrentes: el contenedor periódico y el contenedor esporádico. Su implementación es muy similar ya que ambos se traducen en una tarea cuya única diferencia es precisamente el punto de entrada mencionado en el párrafo anterior. Por este motivo, en el caso del contenedor esporádico es necesario utilizar un objeto protegido o un *Suspension Object* (si no hay transferencia de datos) para implementar el módulo de control. El listado 5.3 muestra el ejemplo de un contenedor planificable periódico, en el que se identifican fácilmente las líneas de código que provienen de los diferentes subsistemas del contenedor. Así, el subcomponente tarea se corresponde directamente con el esqueleto de la tarea Ada. El módulo funcional es representado por la estructura *case*, mientras que el módulo de control, encargado de la invocación de la tarea, en el caso periódico viene representado por la instrucción *delay until* y la actualización que en cada iteración del bucle ajusta el momento en el que la tarea debe ser activada de nuevo.

Tanto el periodo como la prioridad de la tarea son representados por las constantes *Mi_Prioridad* y *Mi_Periodo* respectivamente, cuyos valores son *heredados* directamente de los atributos del contenedor concurrente que da origen a la tarea. En cuanto al valor inicial de la variable *Proxima_Activacion*, se obtiene de un paquete denominado *Global* en el que se almacenan las variables globales, entre las cuales se encuentra el instante de activación inicial.

El listado 5.4 muestra la implementación de un contenedor planificable esporádico, en el que resalta la necesidad de utilizar un mecanismo ligeramente más complejo para implementar el módulo de control que gestiona la activación de la tarea cuando se recibe el evento externo correspondiente. En este caso se ha optado por utilizar un objeto protegido para gestionar la transferencia de control y datos, aunque este mecanismo podría simplificarse ligeramente en caso de no existir transferencia de datos utilizando un objeto del tipo *Suspension Object* en lugar del objeto protegido y una instrucción *Suspend_Until_True* en lugar de la llamada a la entrada del mismo.

Listado 5.3: Implementación de un contenedor planificable periódico

```
package Planificable is
   task Planificable is
      pragma Priority (Mi_Prioridad);
   end Planificable;
end Planificable;

package body Planificable is
   task body Planificable is
      Proxima_Activacion : Ada.Real_Time.Time     := Global.Activacion;
      Periodo : constant Ada.Real_Time.Time_Span := Mi_Periodo;
   begin
      loop
         -- Modulo de control
         delay until Proxima_Activacion;
         -- Modulo funcional
         case Peticion is
            when Valor_1 => Ejecutar_Opcion_1;
            when Valor_2 => Ejecutar_Opcion_2;
               -- ...
         end case;
         -- Modulo de control
         Proxima_Activacion := Proxima_Activacion + Periodo;
      end loop;
   end Planificable;
end Planificable;
```

Listado 5.4: Implementación de un contenedor planificable esporádico

```
package Planificable_Esporadico is
   protected Evento is
      pragma Priority (Mi_Techo);
      entry Esperar_Evento (D : out Data);
      procedure Señalar_Evento (D : in Data);
   private
      Condicion : Boolean;
      Datos     : Data;
   end Evento;

   task Planificable_Esporadica is
      pragma Priority (Mi_Prioridad);
   end Planificable_Esporadica;
end Planificable_Esporadico;

package body Planificable_Esporadico is
   protected body Evento is
      entry Esperar (D : out Data) when Condicion is
      begin
         D := Datos;
         Condicion := False;
      end Esperar;

      procedure Señalar (D : in Data) is
      begin
         Datos := D;
         Condicion := True;
      end Señalar;
   end Evento;

   task body Planificable_Esporadica is
      Datos : Data;
   begin
      loop
         -- Modulo de control
         Evento.Esperar (Datos);
         -- Modulo funcional
         case Peticion is
            when Valor_1 => Ejecutar_Opcion_1;
            when Valor_2 => Ejecutar_Opcion_2;
               -- ...
         end case;
      end loop;
   end Planificable_Esporadica;
end Planificable_Esporadico;
```

5.6. Soporte para particiones

Para facilitar la comprensión, en la sección 5.5 se han mostrado los resultados de las transformaciones a código fuente Ada 2005 de los contenedores a nivel de máquina virtual sin tener en cuenta los elementos de la arquitectura de bandas de prioridad. Partiendo de esta base, a lo largo de esta sección se muestra el código que se ha de añadir para integrar las funciones necesarias impuestas por la arquitectura para lograr el aislamiento temporal efectivo entre particiones.

A pesar de que el código necesario para implementar cada medida se presenta separadamente por razones de claridad, las diferentes opciones son combinables entre sí para obtener un mayor nivel de protección.

5.6.1. Detección simple

El listado 5.5 muestra la implementación del mecanismo de detección simple. El funcionamiento es muy sencillo, mediante una sola llamada al reloj de tiempo de ejecución de la tarea y su comparación con el valor anterior se conoce el tiempo de procesador que la tarea ha consumido en una vuelta completa al bucle principal. Cuando se obtiene dicho valor se compara con el tiempo de ejecución en el peor caso posible (WCET, extraído de los atributos del contenedor). En caso de que haya sido superado se debe ejecutar un procedimiento acorde con alguna de las medidas de contención de fallos propuestas.

En el patrón se ha supuesto que el contenedor planificable sobre el que se implementa este mecanismo es periódico. En caso de ser esporádico no habría ninguna variación más allá de la sustitución de la instrucción *delay* por una llamada a la entrada protegida correspondiente.

No debe olvidarse que el uso de la detección simple ha sido desaconsejado en la evaluación como medida de protección debido a que la latencia entre que se produce el error y se detecta puede ser indefinidamente largo, por lo que este procedimiento sólo debe utilizarse para obtener medidas o con fines estadísticos.

Listado 5.5: Implementación de la detección simple

```
package Deteccion_Simple is
   task Simple is
      pragma Priority (Mi_Prioridad);
   end Simple;
end Deteccion_Simple;

package body Detección_Simple is
   task body Simple is
      Lectura_Actual     : Ada.Execution_Time.CPU_Time;
      Lectura_Anterior   : Ada.Execution_Time.CPU_Time;
      Consumo            : Ada.Real_Time.Time_Span;
      WCET               : constant Ada.Real_Time.Time_Span := Mi_WCET;
      Periodo            : constant Ada.Real_Time.Time_Span := Mi_Periodo;
      Proxima_Activacion : Ada.Real_Time.Time := Global.Activacion;
   begin
      Lectura_Anterior := Ada.Execution_Time.Clock;
      loop
         delay until Proxima_Activacion;
         case Peticion is
            when Valor_1 => Ejecutar_Opcion_1;
            when Valor_2 => Ejecutar_Opcion_2;
            -- ...
         end case;
         Lectura_Actual := Ada.Execution_Time.Clock;
         Consumo := Lectura_Actual - Lectura_Anterior;
         Lectura_Anterior := Lectura_Actual;
         if Consumo > WCET then
            Disparar_Alarma;
         end if;
         Proxima_Activacion := Proxima_Activacion + Periodo;
      end loop;
   end Simple;
end Deteccion_Simple;
```

5.6.2. Detección inmediata

El listado 5.6 presenta el código fuente necesario para llevar a cabo el proceso de detección inmediata. Para ello se hace uso de los temporizadores de tiempo de ejecución incluidos en Ada 2005. Observando el cuerpo de la tarea puede verse que ahora, la primera instrucción arma un temporizador con el WCET de la tarea, cuyo valor viene heredado de los atributos del contenedor padre. El resto del cuerpo de la tarea es idéntico al caso general. No obstante, para utilizar los temporizadores de tiempo de ejecución es necesario añadir otros componentes al paquete, principalmente un procedimiento protegido,

etiquetado en el ejemplo como *Manejar_Timer*, que será el que se ejecute en caso de que el temporizador expire.

Como curiosidad, notar que la instrucción que arma el temporizador de tiempo de ejecución se hace de manera indirecta. Se ha preferido hacerlo de este modo porque permite actuar fácilmente sobre el temporizador desde fuera de la tarea, lo cual, en este momento carece de interés, pero se verá más adelante que facilita la implementación de algunos mecanismos de recuperación de errores, como por ejemplo, el algoritmo de la segunda oportunidad.

Listado 5.6: Implementación de la detección inmediata

```
package Deteccion_Inmediata is
   task Inmediata is
      pragma Priority (Mi_Prioridad);
   end Inmediata;

   protected Manejador is
      pragma Interrupt_Priority (Min_Handler_Ceiling);
      procedure Armar_Timer (Budget : in Time_Span);
      procedure Manejar_Timer (TM : in out Timer);
   end Manejador;
end Deteccion_Inmediata;

package body Deteccion_Inmediata is

   Inmediata_Manejador_WCET  : Timer_Handler := Manejador.Manejar_Timer'Access;
   Inmediata_Id              : aliased constant Task_Id := Inmediata'Identity;
   Inmediata_Timer           : Timer (Inmediata_Id'Access);

   protected body Manejador is
      procedure Armar_Timer (Cuota : in Time_Span) is
      begin
         Ada.Execution_Time.Timers.Set_Handler
           (Inmediata_Timer, Cuota, Inmediata_Manejador_WCET);
      end Armar_Timer;

      procedure Manejar_Timer (TM : in out Timer) is
      begin
         -- Ejecucion de tecnicas de recuperacion
      end Manejar_Timer;
   end Manejador;

   task body Inmediata is
      Proxima_Activacion : Time      := Global.Activacion;
      Periodo            : Time_Span := Mi_Periodo;
      Inmediata_WCET     : Time_Span := Mi_WCET;
   begin
      loop
         Manejador.Armar_Timer (Inmediata_WCET);
         delay until Proxima_Activacion;
         case Peticion is
            when Valor_1 => Ejecutar_Opcion_1;
            when Valor_2 => Ejecutar_Opcion_2;
               -- ...
         end case;
         Proxima_Activacion := Proxima_Activacion + Periodo;
      end loop;
   end Inmediata;
end Deteccion_Inmediata;
```

5.6.3. Cumplimiento del periodo mínimo entre activaciones esporádicas

Hacer cumplir el periodo mínimo entre dos activaciones consecutivas de una tarea esporádica es fundamental para conservar en tiempo de ejecución la validez de los cálculos realizados en el análisis estático. Forzar esta condición resulta muy sencillo tal y como puede observarse en el listado 5.7.

Para conseguir que se respete el intervalo mínimo entre dos activaciones se añaden dos matices al código producido para el caso general de un contenedor planificable esporádico sin protección (listado 5.4). Se trata de una instrucción *delay until* para que, tras concluir su trabajo útil, la tarea quede suspendida hasta que transcurra el mencionado intervalo mínimo (de nuevo, este valor se conoce gracias al atributo del contenedor correspondiente). De esta manera, aunque el dispositivo que dispara la ejecución de la tarea haga llamadas al procedimiento protegido *Señalar*, éstas no tendrán efecto hasta pasado el intervalo mínimo. Para calcular el instante en el que se desea despertar a la tarea (variable *Activación*) se hace una lectura del reloj de tiempo real y se le suma el valor del periodo mínimo entre llegadas. La cuestión clave es, en qué momento debe hacerse la lectura del reloj de tiempo real. Parece claro que no debe hacerse dentro de la propia tarea esporádica ya que nada garantiza que la tarea haya estado bloqueada por otras de mayor prioridad y, por tanto, como resultado se estaría provocando que el tiempo mínimo entre llegadas real fuese mayor que el utilizado en los cálculos teóricos. Por este motivo, se ha decidido hacer la lectura del reloj dentro del procedimiento protegido que levanta la barrera y despierta a la tarea esporádica.

Por otra parte, aunque con esta medida se consiga el efecto deseado de forzar que entre dos activaciones consecutivas transcurra el periodo mínimo especificado, el diseñador del sistema podría entender que este método provoca una pérdida importante de información, ya que se corrige el problema en caso de que se produzca pero no se da ninguna información acerca de la causa. Para corregir esta falta de información, se puede completar la estructura del objeto protegido que regula la llegada de eventos y activa la tarea esporádica tal y como se muestra en el listado 5.8. De esta forma, cuando la tarea esporádica es desbloqueada recibe varios parámetros que le indican, además del momento en el que se disparó el último evento, el número de eventos que se han perdido entre la activación anterior y la actual, así como el intervalo mínimo real entre dos activaciones consecutivas.

Listado 5.7: Cumplimiento del periodo mínimo entre activaciones esporádicas

```
package Planificable_Esporadico is
   protected Evento is
      pragma Priority (Mi_Techo);
      entry Esperar (D : out Data; H : out Time);
      procedure Señalar (D : in Data);
   private
      Condicion : Boolean := False;
      Datos     : Data;
      Hora      : Time;
   end Evento;

   task Planificable_Esporadica is
      pragma Priority (Mi_Prioridad);
   end Planificable_Esporadica;
end Planificable_Esporadico;

package body Planificable_Esporadico is
   protected body Evento is
      entry Esperar (D : out Data; H : out Time) when Condicion is
      begin
         H := Hora;
         D := Datos;
         Condicion := False;
      end Esperar;

      procedure Señalar (D : in Data) is
      begin
         Datos := D;
         Hora := Ada.Real_Time.Clock;
         Condicion := True;
      end Señalar;
   end Evento;

   task body Planificable_Esporadica is
      Activacion          : Time;
      Intervalo_Minimo    : constant Time_Span := Mi_Intervalo_Minimo;
      Datos               : Data;
   begin
      loop
         Evento.Esperar (Datos, Activacion);
         case Peticion is
            when Valor_1 => Ejecutar_Opcion_1;
            when Valor_2 => Ejecutar_Opcion_2;
               -- ...
         end case;
         Activacion := Activacion + Intervalo_Minimo;
         delay until Activacion;
      end loop;
   end Planificable_Esporadica;
end Planificable_Esporadico;
```

Listado 5.8: Mejora de la información en activaciones esporádicas

```
protected Evento is
   entry Esperar (D          : out Data;
                  H          : out Ada.Real_Time.Time;
                  Intervalo  : out Ada.Real_Time.Time_Span;
                  Perdidos   : out Natural);
   procedure Señalar (D : in Data);
private
   pragma Priority (Mi_Techo);
   Condicion            : Boolean := False;
   Datos                : Data;
   Hora                 : Time := Time_First;
   Intervalo_Real       : Time_Span;
   Eventos_Perdidos     : Integer;
end Evento;

protected body Evento is
   entry Esperar (D          : out Data;
                  H          : out Time;
                  Intervalo  : out Time_Span;
                  Perdidos   : out Natural)
     when Condicion is
   begin
      D := Datos;
      H := Hora;
      Perdidos := Eventos_Perdidos;
      Intervalo := Intervalo_Real;
      Condicion := False;
   end Esperar;

   procedure Señalar (D : in Data) is
      Llegada_Actual   : Time;
      Intervalo_Actual : Time_Span;
   begin
      Llegada_Actual   := Ada.Real_Time.Clock;
      Intervalo_Actual := Llegada_Actual - Hora;
      if Condicion then
         -- Se estan perdiendo eventos
         Eventos_Perdidos := Eventos_Perdidos + 1;
         if Intervalo_Actual < Intervalo_Real then
            Intervalo_Real := Intervalo_Actual;
         end if;
      else
         Eventos_Perdidos := 0;
         Intervalo_Real := Intervalo_Actual;
      end if;
      Datos := D;
      Hora  := Llegada_Actual;
      Condicion := True;
   end Señalar;
end Evento;
```

5.6.4. Vigilancia del plazo de respuesta

Para monitorizar si alguna tarea no es capaz de terminar su trabajo antes de que expire su plazo de respuesta se utiliza una implementación basada en los eventos programados de Ada 2005 (ver listado 5.9).

El funcionamiento de este servicio consiste en que cada tarea programe uno de estos eventos para el momento en el que finaliza el plazo de respuesta de su próxima activación. De esta forma, si la tarea consigue terminar el trabajo correspondiente a su activación *n* a tiempo, vuelve a programar el mismo evento, sobreescribiendo su valor, para actualizarlo al instante en el que cumplirá el plazo de respuesta de la activación *n+1*.

Obviamente este mecanismo sólo es válido para el caso de tareas periódicas ya que si hablamos de tareas esporádicas, no es posible saber en un momento dado cuál será el momento en el que se active de nuevo la tarea y, por consiguiente, tampoco se conoce cuál es el instante en el que expirará el plazo de respuesta de la activación *n+1*. En su lugar, ha de utilizarse un mecanismo similar al propuesto en el apartado anterior, de manera que el evento programado no sea fijado en el cuerpo de la tarea (ya podría ser demasiado tarde), sino que la instrucción *Set_Handler* debe ser llamada desde el propio procedimiento protegido que levanta la barrera que permite la activación de la tarea esporádica (ver listados 5.10 y 5.11). Sin embargo, esta acción debe hacerse con sumo cuidado. La instrucción *Set_Handler* no se puede ejecutar sin comprobar antes que el evento programado esté desarmado, ya que se corre el riesgo de sobreesctibir un evento existente, alargando erróneamente el plazo de respuesta de la activación en curso. Por tanto, en caso de que se produzca una llamada al procedimiento protegido *Señalar* cuando el evento programado esté armado, tan sólo se debe anotar el instante en el que se ha producido dicha llamada.

Por otra parte, mientras que en tareas periódicas no hay necesidad de cancelar el evento programado porque simplemente se rearma con el plazo de respuesta de la activación siguiente, en el caso de tareas esporádicas es necesario cancelar el evento programado. Es precisamente en el momento de cancelarlo cuando se debe comprobar si ya se ha producido alguna llamada nueva a *Señalar*, en cuyo caso se reprograma el evento utilizando la variable *Llegada_Actual*, que conserva el instante en el que se produjo la última llamada a *Señalar*, o bien, si no se han producido nuevas llamadas a dicha función, directamente se cancela el evento programado.

Listado 5.9: Implementación de la vigilancia del plazo de respuesta en tareas periódicas

```
package Vigilancia_Deadline is
   task Vigilada is
      pragma Priority (Mi_Prioridad);
   end Vigilada;

   protected Manejador is
      pragma Interrupt_Priority (System.Interrupt_Priority'Last);
      procedure Deadline_Handler (Event : in out Timing_Event);
   end Manejador;
end Vigilancia_Deadline;

package body Vigilancia_Deadline is

   Vigilada_Timing_Event   : Timing_Event;
   Vigilada_Manejador      : Timing_Event_Handler
     := Manejador.Deadline_Handler'Access;

   protected body Manejador is
      procedure Deadline_Handler (Event : in out Timing_Event) is
      begin
         -- Ejecucion de tecnicas de recuperacion;
      end Deadline_Handler;
   end Manejador;

   task body Vigilada is
      Proxima_Activacion : Time := Global.Activacion;
      Periodo            : Time_Span := Mi_Periodo;
      Vigilada_Deadline  : Time_Span := Mi_Deadline;
      Expiracion         : Time_Span;
   begin
      loop
         Expiracion := Proxima_Activacion + Vigilada_Deadline;
         Ada.Real_Time.Timing_Events.Set_Handler (Vigilada_Timing_Event,
                                                  Expiracion,
                                                  Vigilada_Manejador);
         delay until Proxima_Activacion;
         case Peticion is
            when Valor_1 => Ejecutar_Opcion_1;
            when Valor_2 => Ejecutar_Opcion_2;
               -- ...
         end case;
         Proxima_Activacion := Proxima_Activacion + Periodo;
      end loop;
   end Vigilada;
end Vigilancia_Deadline;
```

Listado 5.10: Especificación de vigilancia del plazo de respuesta en tareas esporádicas

```
package Vigilancia_Deadline is
   task Vigilada is
      pragma Priority (Mi_Prioridad);
   end Vigilada;

   protected Manejador is
      pragma Interrupt_Priority (System.Interrupt_Priority'Last);
      procedure Deadline_Handler (Event : in out Timing_Event);
   end Manejador;

   protected Evento is
      pragma Priority (Mi_Techo);
      entry Esperar_Evento (D : out Data);
      procedure Señalar_Evento (D : in Data);
      procedure Cancelar;
   private
      Barrera        : Boolean := False;
      Evento_Armado  : Boolean := False;
      Datos          : Data;
      Llegada_Actual : Time;
      Expiracion     : Time;
   end Evento;
end Vigilancia_Deadline;
```

Listado 5.11: Implementación de la vigilancia del plazo de respuesta en tareas esporádicas

```
package body Vigilancia_Deadline is

   Vigilada_Timing_Event   : Timing_Event;
   Vigilada_Manejador      : Timing_Event_Handler
     := Manejador.Deadline_Handler'Access;
   Vigilada_Deadline       : constant Time_Span := Mi_Deadline;

   protected body Manejador is
      procedure Deadline_Handler (Event : in out Timing_Event) is
      begin
         -- Ejecucion de tecnicas de recuperacion;
      end Deadline_Handler;
   end Manejador;

   protected body Evento is
      entry Esperar (D : out Data) when Condicion is
      begin
         D := Datos;
         Barrera := False;
      end Esperar;

      procedure Señalar (D : in Data) is
      begin
         if Evento_Armado then
            Llegada_Actual := Ada.Real_Time.Clock;
         else
            Ada.Real_Time.Timing_Events.Set_Handler
              (Vigilada_Timing_Event, Vigilada_Deadline, Vigilada_Manejador);
            Evento_Armado := True;
         end if;
         Datos := D;
         Barrera := True;
      end Señalar;

      procedure Cancelar is
      begin
         if Barrera then
            Expiracion := Llegada_Actual + Vigilada_Deadline;
            Ada.Real_Time.Timing_Events.Set_Handler
              (Vigilada_Timing_Event, Expiracion, Vigilada_Manejador);
         else
            Ada.Real_Time.Timing_Events.Cancel_Handler
              (Vigilada_Timing_Event, Cancelado);
            Evento_Armado := False;
         end if;
      end Cancelar;
   end Evento;

   task body Vigilada is
      Datos : Data;
   begin
      loop
         Evento.Esperar (Datos);
         -- Trabajo util...
         Evento.Cancelar;
      end loop;
   end Vigilada;
end Vigilancia_Deadline;
```

5.7. Ejecución de medidas correctoras

Los patrones mostrados a lo largo de esta sección muestran la manera de implementar los diferentes mecanismos de detección de errores temporales utilizando el lenguaje Ada 2005. Una vez conocidos éstos, a continuación se muestran los patrones necesarios para implementar las medidas correctoras explicadas en el capítulo 4.

5.7.1. Esquema propuesto

A grandes rasgos, las tareas se dividen en dos grupos, por un lado se encuentran aquellas que operan en *modo normal*, mientras que por otro lado se agrupan las que deben ser ejecutadas en caso de efectuar un cambio de modo operacional para que el sistema funcione bajo un *modo seguro*. A su vez, las tareas son asignadas a diferentes particiones tal y como se ha explicado en el capítulo 4, por lo que el esquema final, en cuanto a particiones y bandas de prioridad se refiere, se refleja en la figura 5.4, donde pueden verse las aplicaciones que deben ser ejecutadas en el modo normal etiquetadas como *Partición1*, *Partición2* y *Partición3*, mientras que las tareas que se deben ejecutar en el modo seguro se asignan a la banda denominada *Banda de seguridad*.

La figura 5.5 muestra el esquema propuesto para llevar a cabo de manera efectiva un sistema basado en bandas de prioridad. En ella se aprecian los diversos mecanismos de detección y corrección de errores. En primer lugar, el tiempo de ejecución consumido por las tareas está monitorizado por temporizadores de tiempo de ejecución aplicando el algoritmo de la detección inmediata. Por otra parte, eventos programados vigilan el cumplimiento de los plazos de respuesta. Aunque no aparece en la figura, las tareas esporádicas incorporan el mecanismo descrito en el apartado 5.6.3 para evitar activaciones demasiado frecuentes.

Cada uno de los mecanismos mencionados está asociado con un manejador, encargado de llevar a cabo las acciones adecuadas. En caso de que el mecanismo de la detección inmediata dispare una alarma, el manejador se remite al algoritmo de la segunda oportunidad para alargar la cuota si es posible. En caso de que no sea posible conceder esta segunda oportunidad, bien porque se haya consumido la cuota extraordinaria recientemente, bien porque la alarma haya sido disparada por el mecanismo que vigila el cumplimiento del plazo de respuesta, se hace una llamada al procedimiento *Señalar* que activa una tarea esporádica de máxima prioridad encargada de manejar la grave situación en la que se encuentra el sistema. Corresponde al diseñador del sistema decidir qué camino tomar

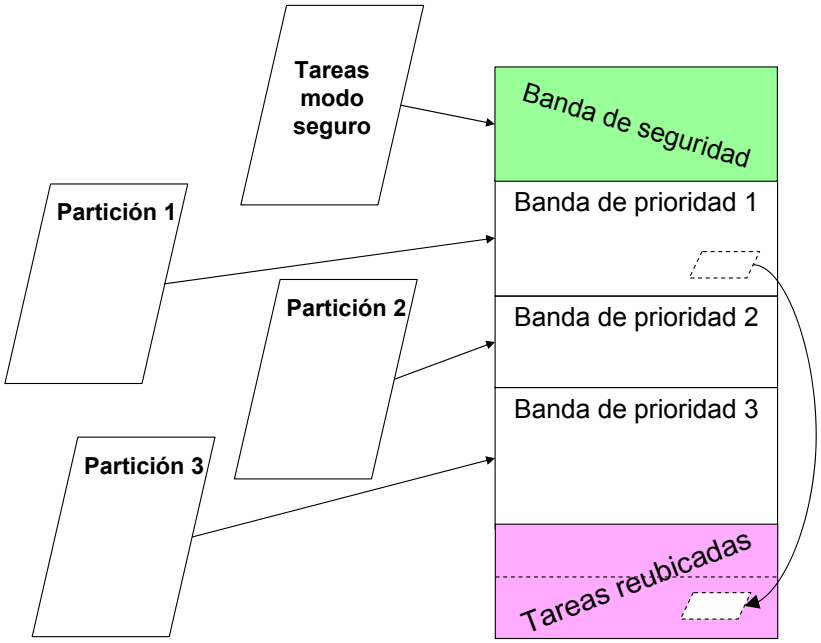

Figura 5.4: División final en bandas de prioridad

en función de la tarea que provoca el mal funcionamiento. Hacer una simple anotación (*Log*) es trivial, mientras que reiniciar el sistema es una opción posible pero que queda al margen de esta tesis.

La última opción que contemplamos consiste en pasar a un modo de operación seguro. Para ello se debe levantar la barrera en la que permanecen suspendidas las tareas etiquetadas como *modo-seguro* desde el comienzo de la ejecución. Debe notarse que el hecho de que estas tareas hayan permanecido suspendidas desde el comienzo en un barrera que es necesario levantar no quiere decir que sean esporádicas; la primera activación puede considerarse excepcional. Una vez que estas tareas entran en el bucle principal de su cuerpo, pueden tener un comportamiento periódico o esporádico exactamente igual que las tareas etiquetadas como *modo-normal*, y cuyos arquetipos se han ido mostrando a lo largo de este capítulo.

Un detalle que consideramos conveniente es añadir una tarea de *background* dentro del modo seguro, es decir, añadir una tarea al grupo seguro, con la prioridad más baja de la banda segura, simplemente para evitar que las tareas de las bandas *normales* vuelvan a entrar en el planificador. Recordemos que si se ha llegado a realizar el cambio de modo operacional es porque el modo *normal* se ha vuelto inestable.

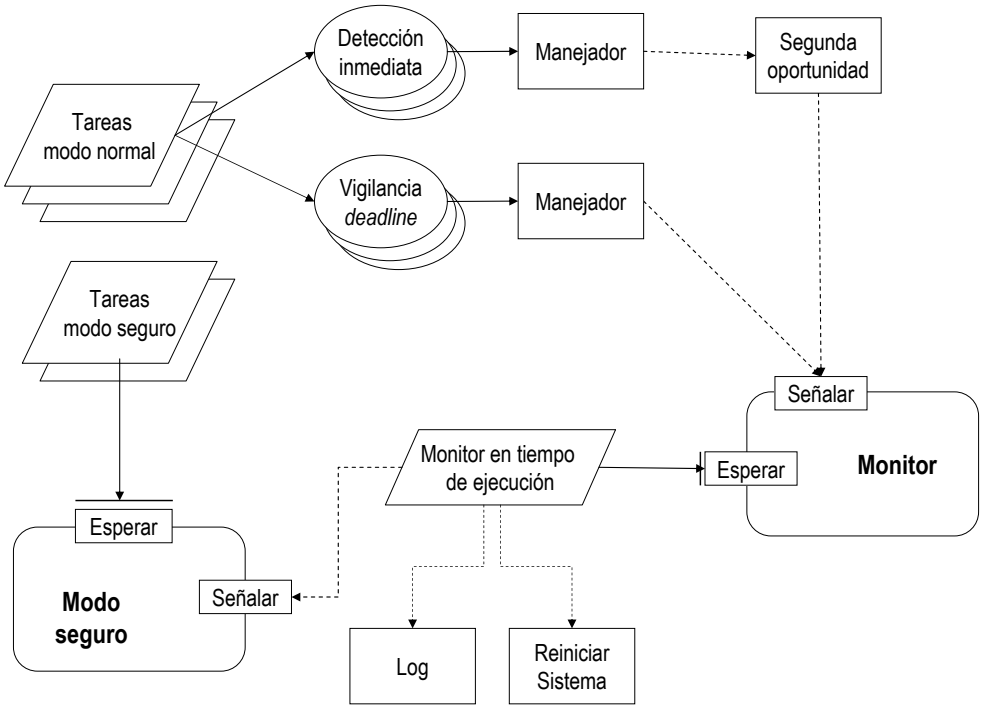

Figura 5.5: Monitorización en tiempo de ejecución

5.7.2. Planificación

El análisis de planificación no sufre grandes cambios por el hecho de implementar más de un modo operacional de la manera propuesta. Al estar las tareas seguras bloqueadas en una barrera desde el principio de la ejecución, no entran en la planificación, es decir, no producen cambios de contexto que rebajarían considerablemente la planificabilidad del sistema. Por tanto, para comprobar que el sistema funciona de manera global, hay que analizar por separado los dos grupos de tareas. Si ambos modos son planificables de manera independiente, también lo será el sistema global.

5.7.3. Implementación

Una tarea perteneciente al modo normal, con todos los mecanismos de protección mostrados en la figura 5.5 se implementa mediante el código mostrado en los listados 5.12, 5.13 y 5.14.

Listado 5.12: Especificación de una tarea monitorizada

```
package Paquete is
   task GNC is
      pragma Priority (Mi_Prioridad);
   end GNC;

   protected GNC_Monitor is
      pragma Interrupt_Priority (Min_Handler_Ceiling);
      procedure Armar_Timer (Budget : in Time_Span);
      procedure Manejar_Timer (TM : in out Timer);
   end GNC_Monitor;

   protected Event_Monitor is
      pragma Interrupt_Priority (System.Interrupt_Priority'Last);
      procedure Manejar_Deadline (Event : in out Timing_Event);
   end Event_Monitor;
end Paquete;
```

Listado 5.13: Cuerpo de una tarea monitorizada (parte I)

```
package body Paquete is

   GNC_WCET_Handler    : Timer_Handler := GNC_Monitor.Manejar_Timer'Access;
   GNC_Id              : aliased constant Task_Id := GNC'Identity;
   GNC_WCET_Violacion  : Timer (GNC_Id'Access);

   GNC_Deadline_Violacion   : Timing_Event;
   GNC_Deadline_Handler     : Timing_Event_Handler
     := Event_Monitor.Manejar_Deadline'Access;

   protected body GNC_Monitor is
      procedure Armar_Timer (Budget : in Time_Span) is
      begin
         Ada.Execution_Time.Timers.Set_Handler
           (GNC_WCET_Violacion, Budget, GNC_WCET_Handler);
      end Armar_Timer;

      procedure Manejar_Timer (TM : in out Timer) is
         Extra : Time_Span := Mi_Cuota_de_Segunda_Oportunidad;
      begin
         if Monitor.Second_Chance.Is_True then
            Armar_Timer (Extra);
            Put_Line ("Segunda oportunidad");
            Monitor.Second_Chance.Set_False;
         else
            Put_Line ("Segunda oportunidad agotada");
            Monitor.Monitor_Release.Signal (Global.Timer_Alarm);
         end if;
      end Manejar_Timer;
   end GNC_Monitor;

   protected body Event_Monitor is
      procedure Manejar_Deadline (Event : in out Timing_Event) is
      begin
         Monitor.Monitor_Release.Signal (Global.Event_Alarm);
      end Manejar_Deadline;
   end Event_Monitor;
```

Listado 5.14: Cuerpo de una tarea monitorizada (parte II)

```
   task body GNC is
      Proxima_Activacion : Time := Global.Activacion;
      Periodo        : Time_Span  := Mi_Periodo;
      GNC_Deadline : Time_Span  := Mi_Deadline;
      GNC_WCET       : Time_Span  := Mi_WCET;
   begin
      loop
         -- Vigilancia de deadline
         Ada.Real_Time.Timing_Events.Set_Handler
           (GNC_Deadline_Violacion,
           (Proxima_Activacion + GNC_Deadline),
            GNC_Deadline_Handler);
         -- Armar timer para deteccion inmediata
         GNC_Monitor.Armar_Timer (GNC_WCET);
         delay until Proxima_Activacion;
         -- Realizar trabajo util
         Proxima_Activacion := Proxima_Activacion + Periodo;
      end loop;
   end GNC;
end Paquete;
```

La tarea monitor encargada de gestionar las alarmas en este ejemplo es sencilla (ver listado 5.15). Simplemente se encuentra bloqueada en una barrera. Cuando uno de los manejadores levanta la barrera, pasa como parámetro un identificador de la causa de la alarma, de manera que el algoritmo elegido por el diseñador del sistema selecciona el procedimiento a ejecutar para salvaguardar la integridad del sistema.

Listado 5.15: Tarea de máxima prioridad que gestiona las alarmas

```
with System;
with Global; use Global;

package Monitor is
   task Monitor is
      pragma Priority (System.Priority'Last);
   end Monitor;

   protected Monitor_Release is
      pragma Interrupt_Priority (System.Interrupt_Priority'Last);
      entry Wait (Al_Type : out Alarm_Type);
      procedure Signal (Al_Type : in Alarm_Type);
   private
      Barrier : Boolean := False;
      Alarm   : Alarm_Type;
   end Monitor_Release;
end Monitor;

package body Monitor is

   protected body Monitor_Release is
      entry Wait (Al_Type : out Alarm_Type) when Barrier is
      begin
         Barrier := False;
         Al_Type := Alarm;
      end Wait;

      procedure Signal (Al_Type : in Alarm_Type) is
      begin
         Alarm := Al_Type;
         Barrier := True;
      end Signal;
   end Monitor_Release;

   task body Monitor is
      Al_Type : Alarm_Type;
   begin
      loop
         Monitor_Release.Wait (Al_Type);
         case Al_Type is
            when Timer_Alarm => Timer_Exhausted;
            when Event_Alarm => Event_Exhausted;
         end case;
      end loop;
   end Monitor;
end Monitor;
```

Capítulo 6

VALIDACIÓN

6.1. Proyectos piloto en ASSERT

Dentro del proyecto ASSERT existe un *cluster* formado por diversas empresas europeas líderes en el sector aeroespacial (Thales Alenia Space, Astrium , EADS Space Transportation, Dassault Aviation y Dutch Space). El objetivo de este *cluster* consiste en integrar, experimentar y validar las soluciones desarrolladas por el resto de los socios del consorcio en sus diferentes *clusters*, así como en asesorarles en la especificación de los requisitos funcionales y operacionales para cada dominio específico.

Para llevar a cabo este objetivo se han definido tres proyectos piloto pensados para probar la validez de estas soluciones en entornos reales. Cada uno de estos proyectos se centra en un área de aplicación concreta:

- HRI: orientado al dominio de los satélites de larga duración sin necesidad de mantenimiento.
- MPC: centrado en los sistemas distribuidos orientados al dominio espacial.
- MA3S: cubre técnicas relativas a los vehículos no tripulados y al dominio de misión crítica.

Durante los primeros meses, el trabajo de los diferentes *clusters* consistió en capturar los requisitos funcionales y operacionales necesarios para desarrollar con éxito los diferentes proyectos piloto. En el caso particular del *cluster* DDHRT, en el que trabajamos,

durante esta fase se definieron las características de la máquina virtual necesarias para poder satisfacer los requisitos expuestos por los proyectos piloto.

Tras una primera fase del proyecto de aproximadamente 18 meses de duración, el *cluster* DDHRT entregó una versión de la máquina virtual, denominada V2, construida conforme a la arquitectura de bandas de prioridad y que ya incorporaba parcialmente funciones de monitorización del tiempo de ejecución consumido por las aplicaciones, gracias al soporte dado por el núcleo de tiempo real basado en ORK que la máquina virtual lleva integrado.

Thales Alenia Space, la empresa que lidera el proyecto piloto HRI, construyó un prototipo con características reales que se ejecutaba sobre la máquina virtual entregada. Es decir, utilizando el modelo de proceso mostrado en el capítulo anterior, se construyó un sistema compuesto por varias aplicaciones cuyo código funcional está embarcado en misiones reales. Siguiendo un proceso de transformaciones verticales se generó un código fuente (del orden de 10.000 líneas de código) legal, admitido por el sistema de compilación GNATforLEON, y soportado por la máquina virtual de ASSERT, generando un archivo ejecutable de aproximadamente 3MB. Finalmente, para la ejecución de este prototipo se utilizó una plataforma *hardware* LEON real.

Durante la revisión intermedia que la Comisión Europea realizó del proyecto ASSERT se presentó este prototipo, que recibió comentarios favorables por parte de los revisores nombrados por dicha Comisión.

Actualmente, las empresas que lideran los proyectos piloto están trabajando en un nuevo prototipo. En esta ocasión, el soporte dado por la máquina virtual será completo, es decir, será conforme a la arquitectura de bandas de prioridad e incluirá el soporte necesario para realizar particiones que garanticen el aislamiento temporal entre las aplicaciones que compartan un mismo nodo.

6.2. Estudio de un caso representativo

Con objeto de llevar a cabo una validación previa, a nivel local, de la arquitectura de bandas de prioridad y los patrones de código fuente mostrados en el capítulo anterior, se ha desarrollado un caso de uso basándonos en las especificaciones propuestas por los socios industriales del proyecto ASSERT, encargados de los proyectos piloto.

6.2.1. Descripción

La descripción del caso de uso a desarrollar es la siguiente.

Sea un subsistema de monitorización en tierra del comportamiento de un satélite en órbita. El subsistema está compuesto por dos aplicaciones que mediante telemetría y telecomandos vigilan el estado de un parámetro y dan órdenes para modificarlo en caso de que sea necesario. Concretamente:

- La aplicación *Control* vigila periódicamente el comportamiento del parámetro *POS*. Dicho parámetro debe mantenerse dentro de un rango de valores $R \in [0, 3)$. En caso de que una lectura del parámetro *POS* devuelva un valor fuera de dicho rango se ejecutará una orden para resetear el valor a cero.
- La aplicación *Actualización* realiza una serie de operaciones internas y como resultado puede incrementar el valor de *POS*.

La aplicación *Control* contiene dos servicios:

- *GNC* es el servicio que desde tierra controla la evolución del parámetro *POS*. Se activa periódicamente cada 2.000 milisegundos.
- *TMTC* es un servicio esporádico que se activa cuando *GNC* detecta un valor de *POS* fuera de rango. Su misión consiste en mandar una orden para devolver a *POS* a su valor inicial. Se considera que este servicio no se activará con un periodo menor de 2.000 milisegundos.

Por su parte, la aplicación *Actualización* tan sólo contiene un servicio, denominado *PRO*, de carácter periódico (con 3.000 milisegundos de periodo) y que modifica el valor de *POS*.

En todos los casos, el plazo de respuesta es igual a los periodos.

Se desea que las aplicaciones se ejecuten en el mismo nodo pero aisladas entre sí para que un error interno a cualquiera de las dos aplicaciones no sea propagado y afecte a la otra. El nivel de criticidad de la aplicación *Control* es mayor que el de *Actualización*.

Por último, se desea que en caso de un mal funcionamiento persistente, el sistema realice un cambio de modo operacional y pase a ejecutar un servicio periódico considerado seguro, que mantenga el sistema en un modo degradado en el que sólo se ejecuten unas

acciones vitales para la supervivencia del sistema. Dicho servicio, cuyo nombre es *Safe* tiene un periodo y un plazo de respuesta igual a 500 milisegundos. Su tiempo de cómputo es de 300 milisegundos.

6.2.2. Transformaciones

El problema descrito puede ser representado mediante contenedores a nivel de aplicación tal y como se muestra en la figura 6.1. La información necesaria para el despliegue viene dada en el último párrafo de la sección anterior, se necesitan dos particiones, una para cada aplicación, dentro de un mismo nodo, de manera análoga a la mostrada en la figura 5.2.

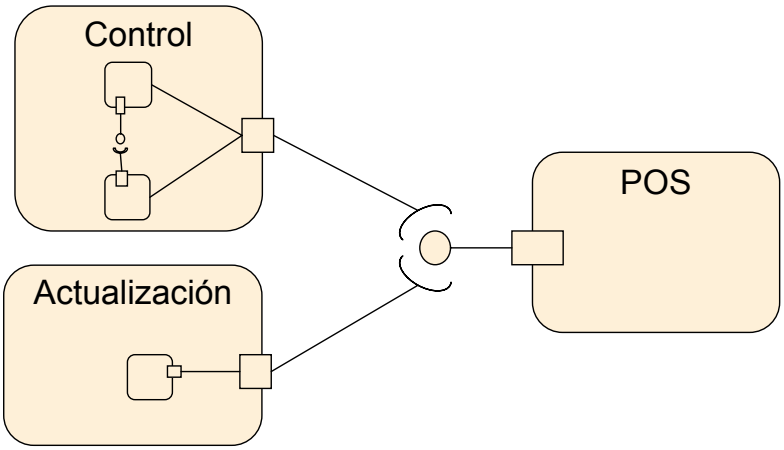

Figura 6.1: Modelo de la demostración

Siguiendo el proceso de transformación vertical, cada uno de los servicios enumerados se transforma en un contenedor concurrente. *GNC* y *PRO* dan lugar a contenedores periódicos mientras que *TMTC* es esporádico. Por otra parte, se genera un contenedor protegido a partir de *PRO*, al que deben acceder los anteriores para consultar o modificar el estado de la variable que comparte el mismo nombre.

Puesto que la información para el despliegue especifica que la criticidad de la aplicación *Control* es mayor, los contenedores concurrentes a los que da lugar esta aplicación deben estar situados en primera instancia en la banda de prioridad superior. Consecuentemente, el contenedor *PRO* se sitúa en la banda de prioridad inferior. El contenedor protegido *POS*, debe tener una prioridad acorde con el protocolo de techo de prioridad

inmediato, por lo tanto, automáticamente se le asigna un valor igual al mayor de todos aquellos contenedores concurrentes que acceden a él.

El cuadro 6.1 muestra los parámetros definitivos que serán traspasados al código fuente, donde *T* representa el periodo o el intervalo mínimo entre llegadas según corresponda, *P* es la prioridad asignada automáticamente, *C* es el tiempo de cómputo, *B* indica el tiempo de bloqueo debido a secciones críticas, *R* es el tiempo de respuesta obtenido en el caso peor para cada tarea mientras que *E* representa la cuota extraordinaria que se puede utilizar en caso de que dispare una alarma el mecanismo de la detección inmediata.

Cuadro 6.1: Parámetros de los contenedores concurrentes

Nombre	**T**	**P**	**C**	**B**	**R**	**E**
GNC	1000	3	300	50	350	450
$TMTC$	2000	2	150	50	500	950
PRO	3000	1	600	0	1350	1200

Puede verse que las tareas correspondientes a los servicios de la aplicación *Control* han sido colocadas en una banda de prioridad superior (prioridades 2 y 3). Dentro de la propia banda se ha utilizado el algoritmo DMS para asignar sus prioridades. Puesto que el análisis de viabilidad da un resultado positivo (todos los tiempos de respuesta son menores que la *deadline* de cada tarea), no es necesario realizar ningún tipo de análisis de sensibilidad en el que se modifiquen parámetros para mejorar los resultados. Lo que sí se ha llevado a cabo es el análisis correspondiente al algoritmo de la segunda oportunidad, dando como resultado que la cuota individual de las tareas se puede aumentar en 450, 950 y 1200 milisegundos respectivamente sin afectar a la viabilidad del sistema.

La tarea que se debe ejecutar en el modo seguro no se ha incluido en el análisis porque al permanecer bloqueada no interfiere en la planificación. Análogamente, el análisis de tiempo de respuesta para el modo seguro es trivial porque este consta de una sola tarea (las tareas pertenecientes al modo de operación normal, mostradas en la tabla no interfieren).

Como resultado final de la transformación se generan seis contenedores concurrentes:

- GNC: implementa el servicio GNC.
- TMTC : implementa el servicio TMTC.
- PRO : implementa el servicio PRO.

- Safe: implementa el servicio seguro *Safe* que ha de ejecutarse sólo en caso de mal funcionamiento persistente.
- Safe_Background: implementa una tarea de sistema que funciona en *background* con un nivel de prioridad dentro de la banda segura para evitar que tras un cambio de modo entren en la CPU las tareas del modo normal.
- Monitor: implementa una tarea del sistema encargada de gestionar las situaciones de mal funcionamiento persistente. En este ejemplo, siempre realiza un cambio de modo.

Además, se obtiene un contenedor protegido, encargado de implementar la variable POS y los servicios que permiten actuar sobre ella.

6.2.3. Resultados

Tras la transformación a código fuente de los contenedores enumerados en el apartado anterior, siguiendo los patrones mostrados en el capítulo 5, se han generado seis tareas, dentro de 12 paquetes (uno para la especificación y otro para el cuerpo de cada una de las tareas), dos paquetes más que incluyen la especificación y el cuerpo del objeto protegido que se genera para dar soporte al contenedor protegido POS y un último paquete en el que se genera el programa principal. Entre todos los paquetes suman aproximadamente 800 líneas de código.

Una vez compilado el código fuente Ada 2005 con el sistema de compilación GNAT-forLEON, que incluye todo el soporte necesario para las funciones que proporcionan aislamiento temporal (relojes y temporizadores de tiempo de ejecución y eventos programados), se obtiene un archivo ejecutable ligeramente inferior a los 700kB de tamaño en memoria.

La figura 6.2 muestra la traza de la demostración. En la parte funcional de la tarea GNC se ha introducido un bucle que aumenta el tiempo de ejecución en cada iteración, de manera que durante las primeras activaciones, la tarea GNC puede finalizar su trabajo sin problemas, sin embargo, tras una serie de activaciones, excede su WCET y el algoritmo de la detección inmediata dispara una alarma. Por ser la primera vez, se hace uso del algoritmo de la segunda oportunidad. Alargar la cuota tiene éxito porque puede comprobarse como la tarea PRO, de menor prioridad, se ejecuta después. No obstante, GNC cada vez necesita consumir más tiempo de procesador por lo que en su siguiente activación vuelve

a saltar la alarma. Al no poder usarse de nuevo el algoritmo de la segunda oportunidad, el fallo pasa a considerarse persistente y se efectúa el cambio de modo, tras el cual, la tarea segura pasa a tomar el control y las demás son desalojadas de la planificación.

```
 TSIM/LEON SPARC simulator, version 2.0.6 (professional version)

 Copyright (C) 2001, Gaisler Research - all rights reserved.
 For latest updates, go to http://www.gaisler.com/
 Comments or bug-reports to tsim@gaisler.com

serial port A on stdin/stdout
allocated 4096 K RAM memory, in 1 bank(s)
allocated 2048 K ROM memory
icache: 1 * 4 kbytes, 16 bytes/line (4 kbytes total)
dcache: 1 * 4 kbytes, 16 bytes/line (4 kbytes total)
section: .text, addr: 0x40000000, size 178608 bytes
section: .data, addr: 0x4002b9b0, size 9220 bytes
read 669 symbols
tsim> resuming at 0x40000000
[ 0.98] - GNC:  0
[ 1.02] - PRO: From 0 To 1
[ 1.98] - GNC:  1
[ 2.98] - GNC:  1
[ 3.98] - GNC:  1
[ 4.32] - PRO: From 1 To 2
[ 4.98] - GNC:  2
[ 5.98] - GNC:  2
[ 6.98] - GNC:  2
[ 7.60] - PRO: From 2 To 3
[ 7.98] - GNC:  3
[ 8.70] - TMTC Activation number: 1 - Level Reset
[ 8.98] - GNC:  0
[ 9.98] - GNC:  0
Segunda oportunidad
[ 10.90] - PRO: From 0 To 1
[ 10.98] - GNC:  1
No es posible utilizar segunda oportunidad
Consumo CPU excedido
[ 11.14] -  Safe Task performing safe actions
[ 11.64] -  Safe Task performing safe actions
[ 12.14] -  Safe Task performing safe actions
[ 12.64] -  Safe Task performing safe actions
[ 13.14] -  Safe Task performing safe actions
[ 13.64] -  Safe Task performing safe actions
[ 14.14] -  Safe Task performing safe actions
[ 14.64] -  Safe Task performing safe actions
[ 15.14] -  Safe Task performing safe actions
[ 15.64] -  Safe Task performing safe actions
[ 16.14] -  Safe Task performing safe actions
```

Figura 6.2: Traza de la demostración

El cuadro 6.2 muestra el número de instrucciones necesarias para ejecutar ciertas acciones significativas como un cambio de contexto y diferentes primitivas relacionadas con los servicios temporales implementados en GNATforLEON, ampliamente utilizados en los patrones de código mostrados en el capítulo anterior. Los valores se han expre-

sado en número de instrucciones ya que el tiempo de ejecución es muy dependiente del *hardware*. La relación ideal es ejecutar una instrucción por cada ciclo de CPU (que en el caso de LEON2 tiene una velocidad de 50MHz), sin embargo, los experimentos reales muestran que el rendimiento habitual es aproximadamente la mitad que el ideal.

Cuadro 6.2: Instrucciones necesarias para ejecutar los diferentes servicios

Operación	**Instrucciones**
$Ada.Real\_Time.Clock$	71
$Ada.Execution\_Time.Clock$	157
$Ada.Execution\_Time.Timers.Set\_Handler$	270
$Ada.Real\_Time.Timing\_Events.Set\_Handler$	270
$Ada.Real\_Time.Timing\_Events.Cancel\_Handler$	400
Cambio de contexto	629
Latencia manejador de Evento Programado)	400
Latencia manejador de Temporizador de Tiempo de Ejecución	415

Capítulo 7

CONCLUSIONES Y TRABAJO FUTURO

7.1. Conclusiones y resultados obtenidos

En el primer capítulo de esta tesis se establecía como objetivo global, diseñar una arquitectura de *software* para sistemas de tiempo real particionados, basada en técnicas de planificación dinámicas, orientada a mejorar la eficiencia obtenida mediante los métodos utilizados actualmente, y que contenga los mecanismos de monitorización necesarios para cumplir con los requisitos de seguridad exigidos en este tipo de misiones.

En esta tesis se ha propuesto una técnica novedosa, denominada *arquitectura de bandas de prioridad*, basada en métodos dinámicos. Dichos métodos deben comenzar a sustituir a otras soluciones muy extendidas actualmente, basadas en técnicas de planificación estática, que lastran la eficiencia de los sistemas de tiempo real críticos embarcados en misiones aeroespaciales.

A pesar de que los métodos de planificación dinámicos mejoran el rendimiento de los sistemas, su uso conlleva tener que afrontar una serie de problemas derivados de la concurrencia, como son el aumento de la complejidad y la necesidad de asegurar aislamiento entre aplicaciones. Sin embargo, a lo largo de este trabajo se han explicado los mecanismos que han de utilizarse para solucionar dichos inconvenientes y que la integridad de un sistema no se vea afectada.

Para conseguir el objetivo global expuesto se ha comenzado por valorar los métodos

utilizados actualmente y sopesar la utilización de nuevas técnicas:

- Se ha realizado una evaluación crítica del estado actual de la técnica, analizando los métodos más relevantes que proporcionan aislamiento temporal, con objeto de destacar y aprovechar sus fortalezas, y a su vez, detectar sus debilidades para minimizarlas. Los resultados de esta evaluación se han expuesto a lo largo del capítulo 3.
- Se ha llevado a cabo un análisis de la nueva revisión del lenguaje Ada, denominada 2005, para comprobar la utilidad de las funciones incorporadas y valorar la posibilidad de diseñar un nuevo mecanismo para proporcionar aislamiento temporal.

Tras realizar este trabajo previo, necesario para sentar las bases de la nueva propuesta se han obtenidos los resultados enunciados a continuación:

- Se ha propuesto una arquitectura basada en el concepto de bandas de prioridad habilitado por el nuevo estándar Ada 2005. Dicha arquitectura presenta varios grados de libertad que permiten configurar el sistema de diferentes formas, por lo que se ha hecho una evaluación de las alternativas posibles en lo referente a:
  - Asignación de prioridades.
  - Elección de políticas de planificación locales.
- Se ha realizado un esquema práctico de integración de las soluciones propuestas dentro del modelo de desarrollo basado en componentes utilizado en el proyecto ASSERT.

Para la validación de la propuesta se ha contado con la colaboración de varios socios industriales de ASSERT (Thales Alenia Space, Astrium y EADS Space Transportation), encargados de diseñar una serie de proyectos piloto con objeto de evaluar el comportamiento de las diferentes soluciones entregadas por otros miembros del consorcio. Dentro del ámbito de esta tesis podemos destacar:

- Thales Alenia Space ha liderado la construcción de un prototipo que utilizaba una versión preliminar de la máquina virtual, la cual incluía las bases de la arquitectura basada en bandas de prioridad y soporte parcial para la monitorización del consumo de tiempo de ejecución por parte de las tareas. En dicha prototipo se ejecutó código real embarcado en misiones espaciales sobre una plataforma *hardware* LEON. La demostración ante los revisores nombrados por la Comisión Europea, encargados de evaluar la calidad del proyecto obtuvo comentarios satisfactorios.

- Se ha desarrollado con éxito un caso de uso, siguiendo las especificaciones propuestas por los socios industriales para comprobar la respuesta de la arquitectura de bandas de prioridad, una vez que la plataforma de ejecución soporta la totalidad de las funciones necesarias para garantizar aislamiento temporal entre aplicaciones.
- Los socios industriales de ASSERT están trabajando en un nuevo prototipo que permita completar la validación de la máquina virtual toda vez que ésta ya implementa todas las funciones requeridas para garantizar aislamiento temporal entre particiones.

Sin embargo, no sólo se han producido resultados dentro del entorno industrial. Los diferentes aspectos relacionados con la arquitectura basada en bandas de prioridad han dado lugar a diversas publicaciones: (Pulido et al., 2006; Pulido, Urueña, Zamorano and de la Puente, 2007; Pulido, de la Puente, Hugues, Bordin and Vardanega, 2007; Pulido et al., 2005; Urueña et al., 2005; Urueña et al., 2007).

7.2. Trabajo futuro

El principal objetivo de esta tesis ha consistido en identificar los problemas que lastran el desarrollo de los sistemas de tiempo real críticos embarcados en misiones aeroespaciales para posteriormente ofrecer soluciones que ayuden a mitigar sus efectos nocivos. Puesto que el objetivo es muy ambicioso, este trabajo no es sino una base para continuar trabajando en la mejora de la eficiencia de este tipo de sistemas.

El trabajo futuro más inmediato consiste en continuar la validación de los resultados obtenidos en esta tesis, ampliando los casos de uso y analizando los resultados obtenidos por las empresas del sector aeroespacial que han comenzado a poner en práctica las técnicas expuestas a lo largo de esta tesis.

El capítulo 4 hace hincapié en el aislamiento temporal entre aplicaciones. El aislamiento espacial ha quedado fuera del ámbito de esta tesis, sin embargo, es el complemento natural para el anterior por lo que una de las líneas de investigación en la que se debe continuar trabajando es en evaluar las diferentes opciones enunciadas en la sección 4.8 y desarrollar el método más conveniente para conseguir garantizar que los espacios de memoria de las diferentes aplicaciones van a ser respetados.

Otra línea de trabajo que debe desarrollarse en el futuro consiste en prestar atención a los temas relacionados con la distribución. Esta tesis se ha centrado solamente en el nivel

local, sin embargo, el crecimiento de los sistemas distribuidos hace necesario ampliar las técnicas desarrolladas en este trabajo para dar servicio a este tipo de sistemas, cada vez más presentes.

Apéndice A

DEFINICIÓN DEL PERFIL DE RAVENSCAR

Los elementos del lenguaje prohibidos por el perfil de Ravenscar son los siguientes:

- Declaraciones de objetos y tareas que no sean a nivel de biblioteca. Es decir, se prohíben las jerarquías de tareas.
- Asignación dinámica o liberación sin supervisión de objetos protegidos y tareas.
- Reencolado.
- Transferencia asíncrona de control.
- Utilizar cualquier forma de la instrucción *select*.
- Utilizar la instrucción *Abort*
- Prioridades dinámicas (a excepción del protocolo de techo de prioridad inmediato que se aplica en los objetos protegidos).
- Utilización del paquete *Calendar*.
- Utilizar retrasos (*Delay*) con valores relativos. Sólo se permiten valores absolutos.
- Declaración de objetos y tipos protegidos fuera del nivel de biblioteca.
- Objetos protegidos con más de un *Entry*.

- Definir *Entries* con barreras que no sean una variable booleana declarada dentro del propio objeto protegido.
- Llamar a un *Entry* que ya tenga una llamada encolada.
- Atributos de tarea definidos por el usuario.
- Modificar dinámicamente la asignación de manejadores de interrupción.

Ada 2005 incluye el perfil en el lenguaje, de manera que se puede dar una directiva al compilador para que detecte cualquier violación del mismo mediante la instrucción:

Pragma Profile (Ravenscar);

Lo cual equivale a todas las directivas mostradas en el listado A.1:

Listado A.1: Directivas equivalentes al perfil de Ravenscar

```
pragma Task_Dispatching_Policy (FIFO_Within_Priorities);

pragma Locking_Policy (Ceiling_Locking);

pragma Detect_Blocking;

pragma Restrictions (
                No_Abort_Statements,
                No_Dynamic_Attachment,
                No_Dynamic_Priorities,
                No_Implicit_Heap_Allocations,
                No_Local_Protected_Objects,
                No_Local_Timing_Events,
                No_Protected_Type_Allocators,
                No_Relative_Delay,
                No_Requeue_Statements,
                No_Select_Statements,
                No_Specific_Termination_Handlers,
                No_Task_Allocators,
                No_Task_Hierarchy,
                No_Task_Termination,
                Simple_Barriers,
                Max_Entry_Queue_Length => 1,
                Max_Protected_Entries  => 1,
                Max_Task_Entries       => 0,
                No_Dependence => Ada.Asynchronous_Task_Control,
                No_Dependence => Ada.Calendar,
                No_Dependence => Ada.Execution_Time.Group_Budget,
                No_Dependence => Ada.Execution_Time.Timers,
                No_Dependence => Ada.Task_Attributes);
```

Bibliografía

Almeida, L. and Pedreiras, P. (2004). Scheduling within temporal partitions: response-time analysis and server design, *EMSOFT '04: Proceedings of the 4th ACM international conference on Embedded software*, ACM Press, New York, NY, USA, pp. 95–103.

Alonso, A. and de la Puente, J. A. (2001). Implementation of mode changes with the Ravenscar profile, *Ada Letters* **XXI**(1). Proceedings of the 11th International Real-Time Ada Workshop.

Amey, P., Chapman, R. and White, N. (2005). Smart certification of mixed criticality systems., *in* T. Vardanega and A. Wellings (eds), *Reliable Software Technologies - Ada-Europe 2005*, Vol. 3555 of *LNCS*, Springer-Verlag, pp. 144–155.

ANS (1983). *Reference Manual for the Ada Programming Language*. ANSI/MIL-STD-1815A-1983.

ARI (2003). *Avionics Application Software Standard Interface — ARINC Specification 653-1*.

Audsley, N., Burns, A., Richardson, M., Tindell, K. and Wellings, A. (1993). Applying new scheduling theory to static priority preemptive scheduling, *Software Engineering Journal* **8**(5).

Audsley, N., Burns, A., Richardson, M. and Wellings, A. (1992). Hard real-time scheduling: The deadline-monotonic approach, *in* W. A. Halang and K. Ramamrithan (eds), *Real Time Programming 1991. Proceedings of the IFAC/IFIP Workshop*, Pergamon Press.

Audsley, N. and Wellings, A. (1996). Analysing apex applications, *rtss* **00**: 39.

Baker, T. P. (1991). Stack-based scheduling for realtime processes, *Real-Time Systems* **3**(1): 67–99.

Baker, T. and Shaw, A. (1989). The cyclic executive model and Ada, *Real-Time Systems* **1**(1).

Barnes, J. (2003). *High Integrity Software: The SPARK Approach to Safety and Security*, Addison Wesley.

Bate, I. and Burns, A. (1997). Schedulability analysis of fixed priority real-time systems with offsets, *Proceedings of the ninth Euromicro Workshop on Real-Time Systems*, IEEE Computer Society Press.

Bernat, G., Colin, A. and Petters, S. M. (2002). Wcet analysis of probabilistic hard real-time systems, *rtss* **00**: 279.

Bini, E., Natale, M. D. and Buttazzo, G. (2006). Sensitivity Analysis for Fixed-Priority Real-Time Systems, *ecrts* **0**: 13–22.

Booch, G., Jacobson, I. and Rumbaugh, J. (2005). *The Unified Modeling Language User Guide, 2nd Edition*, Addison-Wesley.

Booch, G., Rumbaugh, J. and Jacobson, I. (1998). *The Unified Modeling Language User Guide*, Addison-Wesley.

Bordin, M. and Vardanega, T. (2007). Correctness by construction for high-integrity real-time systems: a metamodel-driven approach, *in* N. Abdennadher and F. Kordon (eds), *Reliable Software Technologies — Ada-Europe 2007*, number 4498 in *LNCS*, Springer-Verlag.

Burns, A. (1999). The Ravenscar profile, *Ada Letters* **XIX**(4): 49–52.

Burns, A., Dobbing, B. and Romanski, G. (1998). The Ravenscar tasking profile for high integrity real-time programs, *in* L. Asplund (ed.), *Reliable Software Technologies — Ada-Europe'98*, number 1411 in *LNCS*, Springer-Verlag, pp. 263–275.

Burns, A., Dobbing, B. and Vardanega, T. (2003). Guide for the use of the Ada Ravenscar profile in high integrity systems, *Technical Report YCS-2003-348*, University of York.

Burns, A. and Wellings, A. (1994). HRT-HOOD: A design method for hard-real-time, *Real-Time Systems* **6**(1): 73–114.

Burns, A. and Wellings, A. (1995). *HRT-HOOD(TM): A Structured Design Method for Hard Real-Time Ada Systems*, Elsevier Science, Amsterdam. ISBN 0-444-82164-3.

Burns, A. and Wellings, A. J. (2001). *Real-Time Systems and Programming Languages*, 3 edn, Addison-Wesley.

Buttazzo, G. (2005). Rate monotonic vs. EDF: Judgment day, *Real-Time Systems* **29**(1): 5–26.

Cornhill, D. and Sha, L. (1987). Priority inversion in Ada or what should be the priority of an Ada server task?, *Ada Letters* **7**(7).

Cornhilll, D., Sha, L. and Lehoczky, J. P. (1987). Limitations of ada for real-time scheduling, *IRTAW '87: Proceedings of the first international workshop on Real-time Ada issues*, ACM Press, New York, NY, USA, pp. 33–39.

Davis, R. and Burns, A. (2005a). Hierarchical fixed priority pre-emptive scheduling, *Technical Report YCS-2005-385*, University of York.

Davis, R. and Burns, A. (2006). Resource sharing in hierarchical fixed priority pre-emptive systems, *Proceedings of the 27th IEEE Real-Time Systems Symposium — RTSS 2006*, pp. 257–267.

Davis, R. I. and Burns, A. (2005b). Hierarchical fixed priority pre-emptive scheduling, *RTSS '05: Proceedings of the 26th IEEE International Real-Time Systems Symposium*, IEEE Computer Society, Washington, DC, USA, pp. 389–398.

Davis, R. I., Tindell, K. W. and Burns, A. (1993). Scheduling slack time in fixed priority pre-emptive systems, *IEEE Real-Time Systems Symposium*.

de la Puente, J. A., Ruiz, J. F. and Zamorano, J. (2000). An open Ravenscar real-time kernel for GNAT, *in* H. B. Keller and E. Ploedereder (eds), *Reliable Software Technologies — Ada-Europe 2000*, number 1845 in *LNCS*, Springer-Verlag, pp. 5–15.

de la Puente, J. A., Ruiz, J. F., Zamorano, J., García, R. and Fernández-Marina, R. (2000). ORK: An open source real-time kernel for on-board software systems, *DASIA 2000 - Data Systems in Aerospace*, Montreal, Canada.

de la Puente, J. A. and Zamorano, J. (2003). Execution-time clocks and Ravenscar kernels, *Ada Letters* **XXIII**(4): 82–86.

de la Puente, J. A., Zamorano, J., Ruiz, J. F., Fernández, R. and García, R. (2001). The design and implementation of the Open Ravenscar Kernel, *Ada Letters* **XXI**(1).

Deng, Z. and Liu, J. W.-S. (1997). Scheduling real-time applications in an open environment, *RTSS '97: Proceedings of the 18th IEEE Real-Time Systems Symposium (RTSS '97)*, IEEE Computer Society, Washington, DC, USA, p. 308.

Deng, Z., Liu, J. W. and Sun, S. (1996). Dynamic scheduling of hard real-time applications in open system environment, *Technical report*, Champaign, IL, USA.

Dobbing, B. (2000). Building partitioned architectures based on the ravenscar profile, *Ada Lett.* **XX**(4): 29–31.

Edgar, S. and Burns, A. (2001). Statistical analysis of wcet for scheduling, *RTSS '01: Proceedings of the 22nd IEEE Real-Time Systems Symposium (RTSS'01)*, IEEE Computer Society, Washington, DC, USA, p. 215.

Fuentes, L. and Vallecillo, A. (2004). Una introducción a los perfiles UML, *Novática* **168**.

González-Harbour, M. and Palencia, J. C. (2003). Response time analysis for tasks scheduled under EDF within fixed priorities, *Proceedings of the 24th IEEE Real-Time Systems Symposium*, Cancún, México.

Goodenough, J. and Sha, L. (1988). The priority ceiling protocol: a method for minimizing the blocking of high priority Ada tasks, *Second International Workshop on Real-Time Ada Issues*, ACM SIGAda. Ada Letters, 8(7).

GST (2005). *GSTART: Green Hills Software's Small Tasking Ada Run-Time product*. Available on http://www.ghs.com/products/safety_critical/gstart.html.

Harter, P. (1987). Response times in level-structured systems, *ACM Tr. on Computer Systems* **5**(3).

Int (2005). *Integrity-178B RTOS. Green Hills, Software inc.* Available on http://www.ghs.com/products/safety_critical/integrity-do-178b.html.

ISO (1987). *Reference Manual for the Ada Programming Language*. ISO/8652-1987.

ISO (2000). *Consolidated Ada Reference Manual. Language and Standard Libraries. International Standard ANSI/ISO/IEC-8652:1995(E) with Technical Corrigendum 1*. Available from Springer-Verlag, LNCS no. 2219.

Joseph, M. and Pandya, P. (1986). Finding response times in real-time systems, *BCS Computer Journal* **29**(5): 390–395.

Kuo, T.-W. and Li, C.-H. (1998). A fixed-priority-driven open environment for real-time applications, *Proceedings of the 20th IEEE Real-Time Systems Symposium*.

Leung, J. and Whitehead, J. (1982). On the complexity of fixed-priority of periodic real-time tasks, *Performance Evaluation* **2**(4).

Lipari, G. and Bini, E. (2005). A methodology for designing hierarchical scheduling systems, *Journal of Embedded Computing* **1**(2): 257–269.

Liu, C. and Layland, J. (1973). Scheduling algorithms for multiprogramming in a hard-real-time environment, *Journal of the ACM* **20**(1).

Mazzini, S., D'Alessandro, M., Natale, M. D., Domenici, A., Lipari, G. and Vardanega, T. (2003). HRT-UML: Taking HRT-HOOD onto UML, *in* J.-P. Rosen and A. Strohmeier (eds), *Reliable Software Technologies, Ada-Europe 2003*, number 2655 in *LNCS*, Springer-Verlag, pp. 405–416.

Mok, A. K., Feng, X. A. and Chen, D. (2001). Resource partition for real-time systems, *rtas* **00**: 0075.

Obj (2006). *Unified Modeling Language: Superstructure version 2.1*. OMG document number ptc/06-01-02. Available on http://www.omg.org.

Palencia, J. C. and Harbour, M. G. (1998). Schedulability analysis for tasks with static and dynamic offsets, *RTSS '98: Proceedings of the IEEE Real-Time Systems Symposium*, IEEE Computer Society, Washington, DC, USA, p. 26.

Panunzio, M. (2006). *Teorie e strumenti per l'analisi temporale di sistemi real-time a struttura gerarchica*, PhD thesis, Università degli Studi di Padova.

Panunzio, M. and Vardanega, T. (2007). A metamodel-driven process featuring advanced model-based timing analysis, *in* N. Abdennadher and F. Kordon (eds), *Reliable Software Technologies — Ada-Europe 2007*, number 4498 in *LNCS*, Springer-Verlag.

Pulido, J. A., de la Puente, J. A., Hugues, J., Bordin, M. and Vardanega, T. (2007). Ada 2005 code patterns for metamodel-based code generation, *Proceedings of the 13th International Ada Real-Time Workshop (IRTAW13)*, Woodstock-Vermont, USA. To be published in *Ada Letters*.

Pulido, J. A., Urueña, S., Zamorano, J., Vardanega, T. and de la Puente, J. A. (2006). Hierarchical scheduling with ada 2005, *in* M. G. H. Luís Miguel Pinho (ed.), *Reliable Software Technologies - Ada-Europe 2006*, Vol. 4006 of *LNCS*, Springer Berlin / Heidelberg. ISBN 3-540-34663-5.

Pulido, J. A., Urueña, S., de la Puente, J. A. and Zamorano, J. (2005). Using an architecture description language to model real-time kernels, *in* J. A. de la Puente (ed.), *I Simposio sobre sistemas de tiempo real (CEDI 2005)*, Thomson-Paraninfo. ISBN: 84-9732-448-X.

Pulido, J. A., Urueña, S., Zamorano, J. and de la Puente, J. A. (2007). Handling temporal faults in Ada 2005, *in* N. Abdennadher and F. Kordon (eds), *Reliable Software Technologies — Ada-Europe 2007*, number 4498 in *LNCS*, Springer-Verlag, pp. 15–28.

Rajkumar, R. (1991). *Synchronization in Real-Time Systems: A Priority Inheritance Approach*, Kluwer Academic Publishers, Norwell, MA, USA.

Regehr, J., Reid, A., Webb, K., Parker, M. and Lepreau, J. (2003). Evolving real-time systems using hierarchical scheduling and concurrency analysis, *rtss* **00**: 25.

Robinson, P. J. (1992). *Hierarchical object-oriented design*, Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

SAE (2004). *Architecture Analysis & Design Language (AADL)*. AS5506.

Saewong, S., Rajkumar, R., Lehoczky, J. P. and Klein, M. H. (2002). Analysis of hierarchical fixed-priority scheduling, *Proceedings of the 14th Euromicro Conference on Real-time Systems*, Computer Society, IEEE, Vienna, Austria, pp. 173–181.

Sha, L., Lehoczky, J. and Rajkumar, R. (1986). Solutions for some practical problems in prioritized preemptive scheduling, *IEEE Real-Time Systems Symposium*, IEEE Computer Society Press.

Sha, L., Rajkumar, R. and Lehoczky, J. P. (1990). Priority inheritance protocols: An approach to real-time synchronization, *IEEE Tr. on Computers* **39**(9).

Sha, L., Tarek Abdelzaher, Karl-Erik Årzén, Cervin, A., Baker, T., Alan Burns, Giorgio Buttazzo, Marco Caccamo, John Lehoczky and Mok, A. (2004). Real time scheduling theory: A historical perspective, *Real-Time Systems* **28**: 101–155.

Sprunt, B., Sha, L. and Lehoczky, J. (1989). Aperiodic task scheduling for hard real-time systems, *Real-Time Systems* **1**(1).

Strosnider, J., Lehoczky, J. and Sha, L. (1995). The deferrable server algorithm for enhanced aperiodic responsiveness in hard real-time environments, *IEEE Tr. on Computers* **44**(1).

Taft, S. T., Duff, R. A., Brukardt, R. L., Ploedereder, E. and Leroy, P. (eds) (2007). *Ada 2005 Reference Manual. Language and Standard Libraries. International Standard ISO/IEC 8652/1995(E) with Technical Corrigendum 1 and Amendment 1*, Vol. 4348 of *Lecture Notes in Computer Science*, Springer.

Tokar, J. L. (2003). Space & time partitioning with ARINC 653 and pragma profile, *Ada Letters* **XXIII**(4): 52–54. Proceedings of the 12th International Real-Time Ada Workshop (IRTAW 12).

Urueña, S., Pulido, J. A., Redondo, J. and Zamorano, J. (2007). Implementing the new ada 2005 real-time features on a bare board kernel, *Proceedings of the 13th International Ada Real-Time Workshop (IRTAW13)*, Woodstock-Vermont, USA. To be published in *Ada Letters*.

Urueña, S., Pulido, J. A., Zamorano, J. and de la Puente, J. A. (2005). Adding new features to the open ravenscar kernel, *1st International Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT 2005)*, Palma de Mallorca, Spain.

Urueña, S. and Zamorano, J. (2007). Spatial isolation, *Technical Report 004033.DDHRT_UPM.TN.5*, Universidad Politécnica de Madrid.

Vardanega, T. and Caspersen, G. (2001). Using the Ravenscar Profile for space applications: The OBOSS case, *in* M. González-Harbour (ed.), *Proceedings of the 10th International Workshop on Real-Time Ada Issues*, Vol. XXI, Ada Letters, pp. 96–104.

Vardanega, T., Zamorano, J. and de la Puente, J. A. (2005). On the dynamic semantics and the timing behaviour of Ravenscar kernels, *Real-Time Systems* **29**(1): 1–31.

Vergnaud, T., Hugues, J., Pautet, L. and Kordon, F. (2004). PolyORB: a schizophrenic middleware to build versatile reliable distributed applications, *Proceedings of the 9th International Conference on Reliable Software Techologies Ada-Europe 2004 (RST'04)*, Vol. LNCS 3063, Springer Verlag, Palma de Mallorca, Spain, pp. 106 – 119.

Young, S. (1982). *Real-Time Languages: Design and Development*, Ellis Horwood, Chichester, England.

Zamorano, J., Alonso, A. and de la Puente, J. A. (1997). Building safety critical real-time systems with reusable cyclic executives, *Control Engineering Practice* **5**(7).

Zamorano, J., Alonso, A., Pulido, J. A. and de la Puente, J. A. (2004). Implementing execution-time clocks for the Ada Ravenscar profile, *in* A. Llamosí and A. Strohmeier (eds), *Reliable Software Technologies - Ada-Europe 2004*, Vol. 3063 of *LNCS*, Springer-Verlag. ISBN 3-540-22011-9.

Zamorano, J. and de la Puente, J. A. (2002). Precise response time analysis for Ravenscar kernels, *in* J. Tokar (ed.), *Proceedings of the 11th International Workshop on Real-Time Ada Issues*, Vol. XXII(yy) of *Ada Letters*, ACM Press, pp.ñn–mm.

Zamorano, J. and Ruiz, J. F. (2003). GNAT/ORK: An open cross-development environment for embedded Ravenscar-Ada software, *in* E. F. Camacho, L. Basañez and J. A. de la Puente (eds), *Proceedings of the 15th IFAC World Congress*, Elsevier Press. ISBN 0-08-044184-X.